AD-780 672

A LANGUAGE IMPLEMENTATION SYSTEM

Vernon E. Altman

Massachusetts Institute of Technology

Prepared for:

Office of Naval Research
Advanced Research Projects Agency

May 1974

DISTRIBUTED BY:

NTIS
National Technical Information Service
U. S. DEPARTMENT OF COMMERCE
5285 Port Royal Road, Springfield Va. 22151

AD-780 672

BIBLIOGRAPHIC DATA SHEET	1. Report No. MAC TR-126	2.
3. Recipient's Accession No.		
4. Title and Subtitle: A Language Implementation System	5. Report Date: Issued May 1974	
	6.	
7. Author(s): Vernon E. Altman	8. Performing Organization Rept. No. MAC TR-126	
9. Performing Organization Name and Address: PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY: 545 Technology Square, Cambridge, Massachusetts 02139	10. Project/Task/Work Unit No.	
	11. Contract/Grant No. N00014-70-A-0362-0006	
12. Sponsoring Organization Name and Address: Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Va 22217	13. Type of Report & Period Covered: Interim Scientific Report	
	14.	

15. Supplementary Notes

S.B. and S.M. Thesis, Sloan School of Management, Department of Electrical Engineering

16. Abstracts:

This paper presents the design and implementation of a Language Implementation System (LIS) and investigates the utilization of that system in the development of artificial languages and their associated processors.

The language Implementation System accepts the formal definition of the syntax and semantics of an artificial language, and synthesizes a processor for that language. The parsers (lexical and primary) of the processor are highly efficient Deterministic Push Down Automata (DPDAs) computed from the language's CLR(k) grammar. The CLR(k) (Comprehensive Left to Right, looking ahead k symbols) grammars are defined in the paper, and are shown to include virtually all "practical" artificial languages.

Applications of the Language Implementation System are presented, and the system is shown to be applicable not only to "traditional" artificial languages such as PL/I, Algol, and Lisp, but also to interactive management information/decision system languages.

17. Key Words and Document Analysis. 17a. Descriptors

Language Implementation System, Translator Writing System, Artificial Language, Language Definition, Formal Systems, Formal Semantic Systems, Syntax, Semantics Grammar Complexity Measures, Backus Naur Form, Context Free Grammar, Sentential Form, Language Processor, Interactive Language, LR(k) Hierarchy, LR(k) Grammar, CLR(k) Grammar, Deterministic Push Down Automata, Man-Machine Decision System, Parser, Recognizer, Syntax Directed Error Recovery, Problem Oriented Language, High Level Language, Lexical Analysis, Syntax Analysis

17b. Identifiers/Open-Ended Terms

DDC
RECEIVED
JUL 2 1974
B

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

17c. COSATI Field/Group

18. Availability Statement: Approved for public release; Distribution Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 383
	20. Security Class (This Page) UNCLASSIFIED	22. Price $8.25

FORM NTIS-35 (REV. 3-72)

THIS FORM MAY BE REPRODUCED

USCOMM-DC 14952-P72

383

MAC TR-126

A LANGUAGE IMPLEMENTATION SYSTEM

Vernon E. Altman

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

This thesis was awarded the 1973 E. P. Brooks

Prize for the most outstanding Masters Thesis

Alfred P. Sloan School of Management
Massachusetts Institute of Technology

A LANGUAGE IMPLEMENTATION SYSTEM

by

Vernon Edward Altman

Submitted to the Department of Electrical Engineering and to the Alfred P. Sloan School of Management on May 7, 1973 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science.

ABSTRACT

This paper presents the design and implementation of a particular Language Implementation System (LIS) and investigates the utilization of that system in the development of artificial languages and their associated processors.

The Language Implementation System accepts the formal definition of the syntax (expressed in Backus Naur Form - BNF) and the semantics (expressed in the programming language PL/I) of an artificial language, and synthesizes a processor for that language. The parsers (lexical and primary) of the processor are highly efficient Deterministic Push Down Automata (DPDAs) computed from the language's CLR(k) grammar. The CLR(k) (Comprehensive Left to Right, looking ahead k symbols) grammars are defined in the paper, and are shown to include virtually all "practical" artificial languages. The semantic interpreter of an artificial language is activated for a particular BNF rule whenever a syntactic construct defined by that rule is recognized during the parse of the language's input text.

Applications of the Language Implementation System are presented, and the system is shown to be applicable not only to "traditional" artificial languages such as PL/I, Algol, and Lisp, but also to interactive management information/decision system languages. Furthermore, the processors produced by LIS are not limited to traditional translators, but are also shown to be useful in developing complex Man-Machine decision systems in which they may be viewed as computational dispatchers or structured interfaces between the user and a more complex computational facility.

THESIS SUPERVISOR:	John J. Donovan
TITLE:	Associate Professor of Electrical Engineering
THESIS SUPERVISOR:	David N. Ness
TITLE:	Associate Professor of Management

Acknowledgments

Acknowledgment tradition notwithstanding, the greatest contributor to this thesis has unquestionably been my wife, Mary Lee. Her constant support and encouragement throughout my education at MIT, and more recently with the development of LIS and the production of this thesis, have required personal sacrifices which even I can only begin to appreciate. Mary Lee's contributions are too numerous and personal to pursue here, so let me simply say that her sustained love, patience, and confidence have been profoundly inspirational.

I am deeply grateful to Honeywell Information Systems, Incorporated for the financial support that they have given to the development of LIS over the last two years. A particular note of thanks is due my manager, Mr. Ronald Ham. Ron is thoroughly adept at bridging the difficult gap between promising theoretic developments in computer science and the application of those developments to advanced software engineering. It is largely through Ron's efforts that LIS is gaining acceptance at Honeywell as a viable language development strategy. Special thanks is likewise due Mr. David Ward, a systems analyst of exceptional ability and incredible endurance. Dave has provided an invaluable forum for many of my ideas, and has graciously agreed to assume responsibility for future developments and applications of LIS. I am also grateful to the first users of LIS, Mr. Albert Brown and Mr. Jack Leighton, for their patience and constructive criticisms, to Mr. Bruce Carlson for many useful discussions regarding Multics and various design and implementation strategies, to Mr. William Frink and Mr. Earl Van Horn for their early support of LIS, to Mrs. Pat Lupien for her typing and editing of the LIS User Reference Manual, and to Mrs. Marilyn Barbour for her excellent job on the technical illustrations in this thesis.

My thesis advisors, Professor John Donovan of the Department of Electrical Engineering, and Professor David Ness of the Sloan School of Management, made incalculable contributions to the successful development of this thesis. John's insight into the broad spectrum of formal systems, compiler design, artificial languages, and formal semantic systems were of particular value in structuring the overall objectives of the thesis. Dave's extensive experience in the development of management information/decision systems played a key role in expanding the focus of LIS to include the development and implementation of interactive languages for decision support systems (Appendix E). Professor Malcolm Jones of the Sloan School was most helpful in suggesting ways in which to utilize LIS in the development of general purpose simulation languages. I would also like to thank fellow graduate students Thomas Gearing and Gordon Weekly for their assistance in the development of the Common Base Language Translator. The Translator was developed as a term project for a graduate computer science course, and Appendix A is a modification of the report that we submitted for that project.

I am grateful to Professor Franklin DeRemer of the University of California (Santa Cruz) and to Mr. Wilf Lalonde of the University of Waterloo for their discussions involving the application of LR(k) systems to the development of language implementation system technology. Many of their ideas are incorporated into LIS. I would also like to thank Mr. Sterling Eanes for his helpful discussions on syntax-directed translation strategies.

Finally, I would like to dedicate this thesis to my late parents, Mr. and Mrs. Edward Altman. Their love and confidence will always be remembered.

Vernon E. Altman

Table of Contents

Chapter I

Artificial Language Development
and the
Language Implementation System

Languages die, too, like individuals.... They may be embalmed and preserved for posterity, changeless and static, life-like in appearance but unendowed with the breath of life. While they live, however, they change.

Mario Pei
The Story of Language

I.A Introduction

In this thesis, we present the design and implementation of a particular language implementation system and investigate the utilization of our system in the development of artificial languages and their associated processors. By artificial languages we include, of course, traditional programming languages such as FORTRAN, ALGOL, LISP, PL/I, and COBOL. However, adopting the attitude that an artificial language represents a logical Man-Machine interface, we also include job control languages, end-user language facilities, and interactive management information/decision system languages.

By language processors, we include the compilers and interpreters of such "traditional" languages as cited above. However, here too we expand the traditional terminology to include interactive compilers and complex Man-Machine decision systems in which the language processors may be viewed as computation dispatchers or structured interfaces that permit the end-user to effectively communicate with a more complex computational facility.

The objective of the language implementation system technology (DeRemer first introduced the term - DeR 70) is to provide the language designer with a software capability that addresses itself to the problems of language design and specification, so that by precisely defining the syntax (form) and semantics (meaning) of a particular artificial language, the designer may leave to the language implementation system the task of implementing the processor for his language. This is indeed an ambitious objective and one that has not yet been fully realized. However, substantial advancements have been made in the automation of the syntactic recognition processes for artificial languages, and in the subsequent discussions we present the most significant of these by way of its incorporation into the design of our Language Implementation System (LIS). Furthermore, we define a framework for associating these recognition processes with current and potential advancements in formal semantic systems so that the Language Implementation System may have an acceptable expectation of successfully evolving towards the achievement of the objective previously set forth.

As a first approximation, the motivation for the development of language implementation systems may be seen as evolving out of a natural desire to make the development of processors for traditional languages more flexible, efficient, well-structured, and reliable than has typically been the case with manual or semi-automatic techniques. However, it is part of the present thesis that a sufficiently comprehensive language implementation system will not only facilitate the development of traditional language processors, but will also support the development of languages and processors for special purpose end-user facilities and Man-Machine decision systems in which a primary consideration is the ability of such systems to adapt to an ever changing problem environment. Thus, to the extent that the domain of feasible problem environments to which such computation may be successfully applied is a function of the supporting language facilities, it is our belief that language implementation systems such as our own will play a significant role in the expansion of this domain.

In the remainder of this chapter, we address several topics of an introductory nature. In Section I.B, we present a representative model of a processor for an artificial language and use this model to define the general functional capability of the Language Implementation System. In Section I.C, we get more specific and identify the system objectives and design criteria of LIS. In Section I.D, we suggest the contributions made by this thesis to the development and application of language implementation system technology. In Section I.E, we give the reader a brief overview of our approach in the remainder of the thesis.

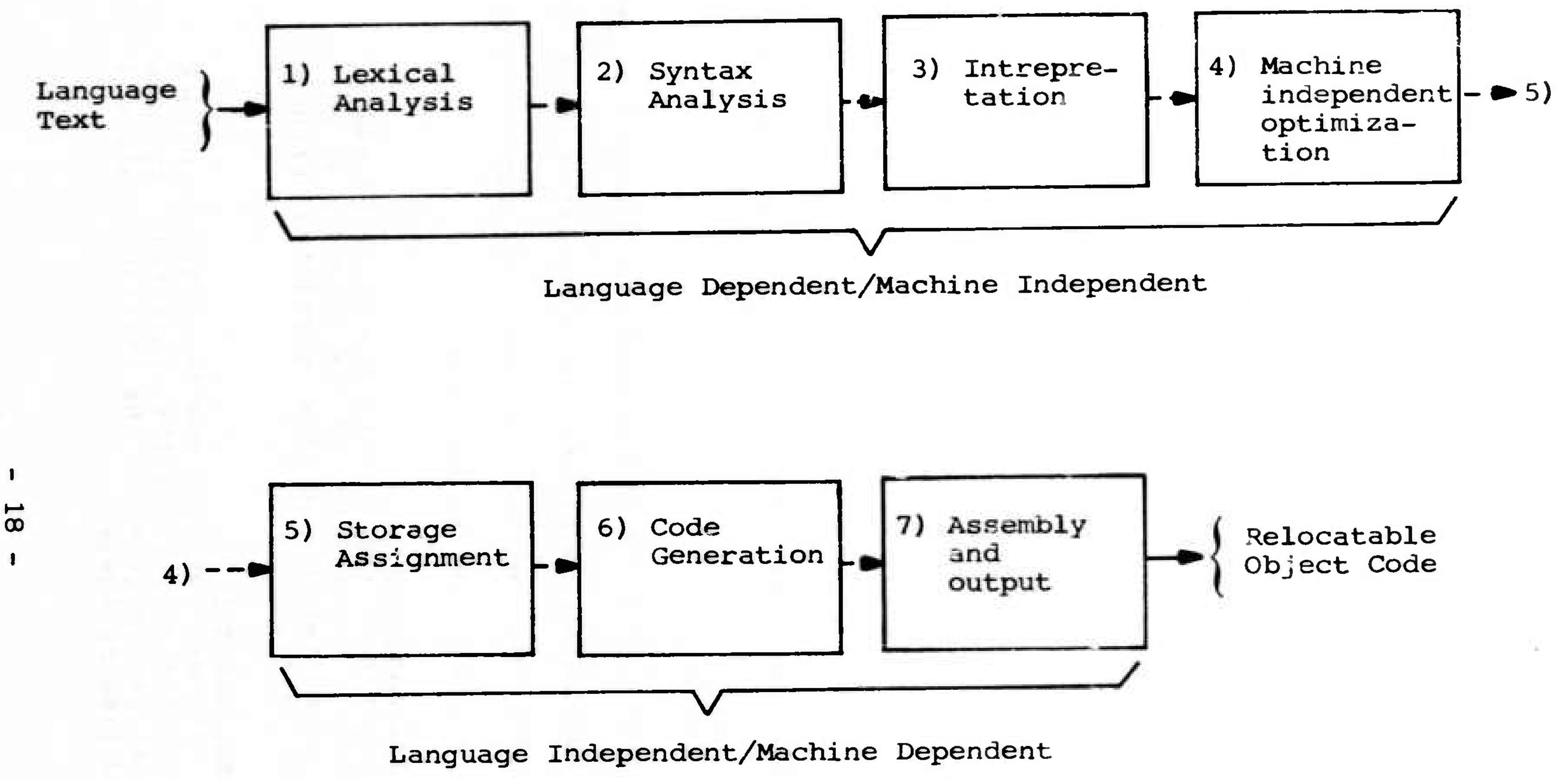
I.B Language Processing Models and the Language Implementation System

In this section, we present a particular language processing model and use that model to more clearly identify the basic objectives of the Language Implementation System. The model that we choose is the classical seven phase compiler model as presented, for example, by Donovan (Don 72). We use the model of a compiler because most of the important issues of language processor design arise in the context of a compiler. As previously indicated, we shall subsequently discuss the application of LIS to other language processors, so that the present discussion is not meant to imply a particular limitation on the system's applicability.

The model that we use is given in Figure I.1, and is strictly a functional model, as the associated data bases have been omitted. Within the model, the functions of the phases may be identified as follows:

1. Lexical Analysis
   The process of recognizing the basic elements, or lexical constructs, of the language as they are encountered in the submitted program (Input Text).

2. Syntax Analysis
   The process whereby the syntactic structure or phrase structure of the submitted program is analyzed in terms of the lexical constructs recognized by Phase 1.



Functional Model of a Compiler

Figure I.1

3. Interpretation
   The process of associating semantics, or meaning, with the phrase structure as analyzed in Phase 2. The semantic interpretation is in the form of an intermediate target language representation and a set of data bases.

4. Machine Independent Optimization
   The process whereby machine independent optimizations are performed on the intermediate target language representation so as to produce a more efficient semantic representation of the submitted program.

5. Storage Assignment
   The process of determining the storage allocation for the program's data elements and the compiler generated data elements.

6. Code Generation
   The process of generating the assembly code representation of the submitted program, with symbolic references.

7. Assembly and Output
   The process of resolving symbolic references and generating the machine language representation of the submitted program.

As indicated in Figure I.1, the first three phases are largely language dependent and machine independent, whereas the subsequent phases are largely language independent and machine dependent. Thus, the theory would imply that for a given machine, one need only be concerned with the first three phases when implementing a new language, and conversely, with only phases four through seven when transporting an existing language to a new computing environment. Of course, such idealizations have rarely been realized.

Given this model, we may now begin to identify more clearly the basic objectives of LIS. Stated most simply, the objectives of LIS are to automate completely the lexical analysis and syntax analysis phases, while providing a convenient framework for associating semantic interpretation with the syntactic constructs of the language being implemented (Phase 3 - Interpretation).

In the next section, we identify the specific objectives and design criteria of LIS.

I.C Objectives and Design Criteria of the Language Implementation System

The objectives of the Language Implementation System are to facilitate the design of artificial languages and the implementation of their processors by:

1. Providing analysis procedures capable of detecting and identifying syntactic ambiguity, as well as for verifying certain characteristics required of well structured artificial languages.

2. Providing the capability to automate completely the implementation of the syntactic recognition processes of artificial languages satisfying the criteria implied above.

3. Providing a convenient structure within which semantic interpretation may be associated with the syntactic constructs of those languages whose recognition processes have been automated by the system.

With respect to the first two objectives, the system should be equally effective when used by either language designers or experienced processor implementers. Since the automation procedures take their input from a formal specification of the syntax of the particular language (expressed in Backus Naur Form), no particular experience with processor implementation should be necessary, or even useful. However, since formal semantic systems have not yet been developed to the point where they are both sufficiently general and efficient as to be incorporated into commercially marketable language processors, our approach with regard to objective three must be to provide a flexible framework within which the designer/implementer may employ the capabilities of a general purpose programming language in associating semantic interpretation with the constructs of his language.

The following design criteria were established in support of the above objectives:

1. The system must be capable of handling a wide variety of programming languages. In the development of new languages, the system should serve as a design tool by pointing out areas of syntactic ambiguity and complexity, since these will be bothersome to both implementer and user. In its application to existing languages, the system should be sufficiently general as to admit the representation of most of these in such a way as to require little or no rework of the language's existing syntax.

2. The syntactic recognizers produced by the system must come with an "error-free" guarantee. That is, given that the system has claimed to have generated a recognizer from a particular specification, we require that:

   a. The recognizer always parse input text generated by the given specification. Since we also require that the system not generate recognizers for ambiguous languages, we therefore know that the resulting parse will be the only parse.

   b. The recognizer always reject input text not generated by the given specification.

   Satisfying this criterion implies that during processor development, all processing errors will, *a priori*, be the result of errors in the semantic specification.

Of course, the specification itself may be in error in the sense that it generates text that the designer "did not have in mind", and although the system will not be able to handle this problem directly, it must be designed in such a way as to allow the user to introduce change into his language in a convenient and timely manner. This requirement leads to the next design criterion.

3. The syntactic meta-language that constitutes the input to the system must be palatable to the system user. Although this characteristic is somewhat difficult to define precisely, it carries with it the following notions:

   a. Non-procedural (independence as to the ordering of the meta-language rules within a particular definition).

   b. Free format (within the meta-language rules).

   c. The ability to handle long (many character) symbols, so as to allow the user to be descriptive in his naming conventions.

   d. A minimal number of reserved symbols.

   In addition, we require that the (system acceptable) syntax specification be such that the same document may appropriately be used to define the syntax of a language to the users of that language. Because of this requirement, we insist that the syntactic meta-language be high-level and resemble, as closely as possible, the most universally accepted meta-language currently found in the computer literature.

4. Procedures must exist to provide for syntax directed error detection, reporting, and recovery when illegal text is encountered during syntax analysis of programs written in the specified language. Although these procedures must be general enough to report and recover from (continue parsing) most context-free syntax errors, a facility must

also be provided to allow the language designer/implementer to substitute his own reporting and recovery procedures in those cases in which the general capability is found to be inappropriate.

5. The syntactic recognizers produced by the system must be _time efficient_ in the sense that their speed of recognition for a particular language must be comparable to, or faster than, alternate methods of implementation for the same language.

6. The syntactic recognizers produced by the system must be _space efficient_ in the sense that the space required for their representation must be comparable to, or less than, alternate methods of implementation for the same language.

7. The structure chosen for language definition must provide a convenient framework for the specification of semantics. In particular, the semantics of a particular syntactic construct should be expressible in-line with the syntax of that construct.

8. The error messages delivered by the system when processing a particular definition should be precise and relative to the input specification. This will allow language debugging to proceed on-line in most instances.

By specifying the above design criteria, we identify, in a general way, the fundamental capabilities of the Language Implementation System. With the exception of the syntax directed error handling capability (which has been designed), all of the criteria have been met in the current implementation of the system.

I.D Thesis Contributions

The theoretic foundation of the Language Implementation System is derived from the work by Knuth (Kn 65) and DeRemer (DeR 69) on LR(k) systems. Knuth developed the theoretic concepts of left to right translation when he defined the LR(k) grammars, although it was not until DeRemer's work that LR(k) systems were seen as having high potential in the area of "practical" language processors. These two works are now regarded as classics in the application of automata theory to language translation, and are strongly recommended to the reader demanding a rigorous treatment of the theoretic foundation of LIS.

The purpose of this thesis is not to go over the theoretic development of LR(k) systems to any significant extent. Rather, we believe that our contributions to the development of language implementation system technology may be identified as follows:

1. The definition of a subset of the LR(k) grammars, called the Comprehensive LR(k) grammars (CLR(k)). We call the grammars "comprehensive" because they include virtually all of the "practical" grammars with which we have come in contact.

2. The design and implementation of a commercially viable language implementation system based on the CLR(k) grammars. LIS is commercially viable in the sense that Honeywell has found it appropriate to use the system, as implemented on Multics, to generate

parsers for languages to be implemented on other computers.

3. The application of LIS to the development of various artificial languages and their associated processors.

I.E Thesis Approach

As already indicated, it is the intent of this thesis to complement the work of Knuth and DeRemer by developing a well-engineered, commerciallv viable language implementation system based on LR(k) techniques, and to investigate the utilization of our svstem in the development of artificial languages and their associated processors. We are in an enviable position, with respect to LR(k) systems, of having a well developed theoretic base. However, our approach in the development of the fundamental algorithms of LIS in Chapter III is to present the notions of finite state machines and deterministic push down automata on what essentially reduce to intuitive notions. That is, given the luxury of a well developed theory, we choose to bypass that theory and appeal to the reader's basic analytic skills, often using examples to illustrate the algorithms.

Even with our emphasis on design and application, however, there remains a moderate amount of terminology and fundamental conceptual material that is essential to our subsequent discussions. This is presented in the first part of Chapter II. Also in Chapter II, we describe the basic structure of LIS from the point of view of artificial language/processor development, and indicate the way in which the fundamental algorithms to be presented in Chapter

III are combined to achieve the functional capability of LIS.

In Chapter III, we develop the fundamental algorithms of LIS.

In Chapter IV, we cover a variety of issues, including the efficiency of the parsers produced by LIS, the design of our syntax directed error handling procedures, the hierarchy of LR(k) systems and how the CLR(k) grammars relate to that hierarchy, and a discussion of some of the more significant applications of LIS to date. Chapter IV concludes the main portion of the thesis, and in the last sections of the chapter we summarize our work and suggest areas of future investigation that can be expected to have significant impact on language implementation system technology.

The thesis is structured so that by reading Chapters I through IV, the reader can grasp the essential subject matter that we are attempting to present. In the appendices, we present many of the details of the system design, implementation, and applications that are likely to be of interest only to individuals actively engaged in the development and application of language implementation systems. Appendix A is the LIS User Reference Manual, which describes LIS from the point of view of the language designer/implementer. In Appendix B, we give a macro description of the design and implementation of LIS, complete with flowcharts of the major procedures and descriptions of the major data structures. In Appendix C, we give an application of LIS to the implementation of a translator from a block structured language into Dennis's Common Base Language (a particular formal semantic system). In Appendix D, we give the syntax of PL/I from which we generated CLR(1) lexical and primary parsers. In Appendix E, we give an example of the application of LIS to the development of an interactive decision support system.

Chapter II

Fundamental Language Concepts and the Structure of the Language Implementation System

II.A Introduction

As indicated in Chapter I, this thesis is oriented towards the design and application of our Language Implementation System, as opposed to a rigorous development of its underlying theory. Even with this design and application orientation, however, there remains a moderate amount of terminology and fundamental conceptual material that is essential to our subsequent discussions. In Section II.B, we give a treatment of this material and acknowledge that most of this treatment is from DeRemer's thesis (DeR 69).

In Section II.C, we present the structure of the Language Implementation System, and discuss its utilization in the development of artificial languages and their associated processors. In addition, we describe the way in which the fundamental algorithms presented in Chapter III are combined to achieve the functional capability of LIS.

Preceding page blank

II.B Fundamental Concepts and Terminology

We begin by defining terms and notation. We assume the reader is familiar with the properties of symbols, strings of symbols, languages, finite state machines (FSMs), formal grammars, and deterministic push down automata (DPDAs).

A context-free grammar (CFG) is a quadruple (Vt, Vn, S, P) where Vt is a finite set of symbols called terminals, Vn is a finite set of symbols distinct from those in Vt called non-terminals, S is a distinguished member of Vn called the starting symbol, and P is a finite set of pairs called productions. Each production is written A->w and has a left part A in Vn and a right part w in V* where V = Vn U Vt. V* denotes the set of all strings composed of symbols in V, including the empty string.

Without loss of generality, we conventionalize that (i) the productions are arbitrarily numbered from 0 to s, and (ii) the zero-th production is of the form S -> ¦-S′-¦, where S′ is sort of a subordinate starting symbol, and S and the "pad" symbols ¦- and -¦ appear in none of the other productions. In addition, we associate the symbol, #p, with the p-th production, so that the production may be conceptually written as (p) A->w #p. The #-symbols are alternatively called apply symbols, and refer to the application of the associated production.

The following is an example of a context-free grammar:

G = (((,),i,↑,+,¦-,-¦), (S,E,T,P), S, P)

where P consists of the following productions:

(0)	S -> ¦- E -¦	(4)	T -> P
(1)	E -> E + T	(5)	P -> i
(2)	E -> T	(6)	P ->(E)
(3)	T -> P ↑ T		

If A->w is a production, an immediate derivation of one string a = vqb from another a′ = vMb is written a′->a. We say that a is immediately derivable from a′ via application of the production M->q to a particular occurrence of M in a′. The transitive completion of this relation is a derivation and is written a′->*a, which means there exists strings a0,a1,...,an such that a′ = a0->a1->...->an = a for $n \geq 0$. A right derivation, written a′->r*a, is one in which for i = 1,2,...,n each a(i) is immediately derivable from a(i-1) via application of the rightmost non-terminal in a(i-1). We choose the right derivation as our canonical derivation.

A terminal string is one consisting entirely of terminals. A sentential form is any string derivable from S. A sentence is any terminal sentential form. The language L(G) generated by G is the set of sentences: i.e., L(G) = (n∈Vt* ¦ S->*n). A right sentential form, which we choose as our *canonical form* is any string canonically derivable from S.

Let M->q be the p-th production of a CFG, and let a′=vMb and a=vqb be canonical forms such that there exists a canonical derivation S->r*a′->a. Then vq#p is a *characteristic string* of a.

Loosely speaking, a *parse* of a string is some indication of how that string was derived. In particular, a *canonical parse* of a sentential form, a, is the reverse of the sequence of productions (or equivalently, the numbers thereof) used in a canonical derivation of a. We refer to the action of determining a parse as *parsing*, the determination constitutes a *grammatical analysis*, and a parsing algorithm is called a *parser*.

For our purposes, a *Finite State Machine* (FSM) may be considered to be a set of *states* and *transitions* from those states. Each *transition* has an associated *transition symbol* which defines an exit from the state to which the transition belongs. The symbol may be a terminal symbol, a non-terminal symbol, or a #-symbol. In the first two cases, the state transition also has an associated *destination state*. The *destination state* identifies the state that is entered upon exiting the original state on the transition symbol. A state having only transitions on terminal and non-terminal symbols is called a _read state_. A state having a single transition, that transition being on a #-symbol, is called an _apply state_. A state having more than one transition, at least one of which is on a #-symbol is called an _inadequate_ state.

A series of transitions leading through an FSM from state N1, to state N2... to state Nk is called a _path_ from N1 to Nk. Every such path _spells out_ a unique string of input symbols (i.e. an _input string_) in the obvious way. An FSM _accepts_ a given string T if and only if there exists at least one path that begins at a (specially marked) starting state, spells out T, and ends at a (specially marked) terminal state. The set of all strings accepted by an FSM is referred to as the set that is _recognized_ by that FSM.

States M and N are said to be _equivalent_ if and only if for each input string T spelled out by some path from M (N), such that the path also spells out the string T′, there exists a path from N (M) which spells out the same two strings T and T′, respectively.

An FSM is said to be _deterministic_ if and only if it has a single starting state and from each state there is at most one transition under each distinct input symbol;

otherwise, it is said to be nondeterministic. A deterministic FSM is said to be reduced if and only if every state is accessible from the starting state, some terminal state is accessible from every state, and no two states are equivalent.

A Characteristic Finite State Machine (CFSM) of a context-free grammar G is a reduced, deterministic FSM which recognizes the set of characteristic strings of G.

We often think of a deterministic FSM as a physical machine, rather than as an abstract model, and this leads to the following terminology. To determine if a given FSM accepts a given string T, we say we initialize the machine (i.e. start it in its starting state), apply it to T, and determine if T takes the machine through a sequence of states to a terminal state. The machine is said to read the symbols in T from an input tape, to enter first one state and then the next, and to output symbols onto an output tape. If after reading the last symbol of T the machine outputs a "1", then it accepts T. However, if at that time it outputs a "0", it does not accept T. The machine stops reading whenever it enters a state with no transition under the next symbol to be read.

A Deterministic Push Down Automaton (DPDA) is a machine that consists of an input tape, an output tape, a finite control and a push down stack.

The finite control can be thought of as a program consisting of instructions pertaining to the reading of symbols from the input tape and the outputting of symbols to the output tape, the storage, interrogation, and removal of items on the stack, and jumps from one point in the program to another. The control can be represented by a transition graph whose nodes are called states, and whose labelled arrows are called transitions.

Each state represents a point in the program which can be jumped to, and it has a name which is given inside the node. There is a unique starting state and a unique terminal state.

Each transition implies one of four kinds of instructions, the interpretations of which are indicated next. If the machine enters state N having a transition to state M, then, if the label of the transition is (1) a symbol s, the machine reads the next symbol, and if the symbol read is s, it then enters state M, (2) "push i", the machine pushes the item i on the stack and then enters state M, (3) "pop n, out p", the machine pops the top n items off the stack, outputs p, and then enters state M, or (4) "top i", the machine compares item i with the top item on the stack, and if they are the same, it enters state M.

The following two conditions are sufficient to guarantee determinism: (1) any state having a transition under either "push i" or "pop n, out p" may have no other transitions, and (2) any other state must have either every transition under a symbol, or every one under "top i" for some item i.

The initial configuration of a DPDA is as follows: It is started in its starting state with the input string on its input tape, with its input head (reading device) over the leftmost symbol of the input string, and with its stack empty. The final configuration is: the input head one place to the right of the rightmost symbol of the input string, the stack empty, and the machine in its terminal state.

The similarity of DPDAs and FSMs is emphasized if we note that a DPDA which never uses its stack is equivalent to some FSM. This leads us to think of a DPDA, then, as being based on some FSM. We think of this FSM as reading symbols, as usual, but interspersed between some of the reads are some "bookkeeping" operations involving the stack, and these operations effect some of the state changes of the FSM.

As we acknowledged, most of the above discussion is from DeRemer. We now relate some of the above concepts and terminology to their particular role in the development of the fundamental algorithms of LIS.

LIS accepts the Backus Naur Form (BNF) as a syntactic meta-language for representing context-free grammars. A BNF specification is composed of BNF rules of the form A ::= B !, where A is the left part of the rule, and B is the right part of the rule. The left part of the rule is the non-terminal that is defined in the rule. The right part of the rule is composed of one or more alternative definitions of the left part, separated by the symbol ";". The alternatives of the right part, in conjunction with the non-terminal left part, correspond to the productions of the context-free grammar. Thus, a BNF rule with n alternative right parts corresponds to n productions, and we often use the term alternative and production interchangeably. In a BNF specification containing n rules, the rules are arbitrarily numbered from 1 to n. In order to uniquely identify the alternatives (productions) of the specification, we also number the alternatives of each rule from one to the number of alternatives in the rule. Thus, the j-th alternative of the i-th rule is uniquely identified as (i: j).

The non-terminal and terminal symbols of a BNF specification are represented in the following way:

1. The non-terminal symbols are represented as character strings enclosed in angle brackets (e.g. "<" character string ">").

2. The terminal symbols are all character strings of the specification excluding non-terminals and the character strings "::=", "|", "!", "<", ">", blank, tab, new-line, and new-page, unless these character strings are "escaped" (see Appendix A).

The goal symbol of the BNF specification is either <primary_non_terminal> or <lexical_non_terminal>, depending on whether the specified grammar is primary or lexical, respectively. The lexical grammar defines the structure of the basic elements of the language (e.g. <identifier>, <integer>) in terms of the legal terminal characters of the language. The primary grammar defines the sentential forms of the language in terms of these basic lexical constructs. In those language definitions in which both lexical and primary grammars are specified, both <primary_non_terminal> and <lexical_non_terminal> must be defined. A more extensive discussion of the structure and format of the LIS Backus Naur Form is given in Appendix A.

The previously defined context-free grammar, G, can be expressed in BNF as follows:

```
(1)  <primary_non_terminal> ::= |- <E> -| !
(2)  <E> ::= <E> + <T> |
             <T> !
(3)  <T> ::= <P> ↑ <T> |
             <P> !
(4)  <P> ::= i | ( <E> ) !
```

In the previous discussion on context-free grammars the term _language_ was formally defined. While we will maintain this formal definition, we will also use the term (as well as the term _artificial language_) in an informal sense in which we include the semantics associated with the syntactic constructs. Thus, we refer to the language (syntax and semantics) PL/I, the language Algol, and languages that constitute the logical interface between the end-user and an interactive computational facility, i.e. interactive languages.

The term _parser_ will be retained as previously defined, and we shall differentiate between lexical parsers and primary parsers where such differentiation is appropriate. We also make occasional reference to the term _recognizer_, which for our purposes is synonymous with the term parser.

II.C The Structure of the Language Implementation System

The fundamental structure of the Language Implementation System is indicated in Figure II.1. Language development resolves into two interacting phases, Processor Generation and Processor Execution.

II.C.1 Processor Generation

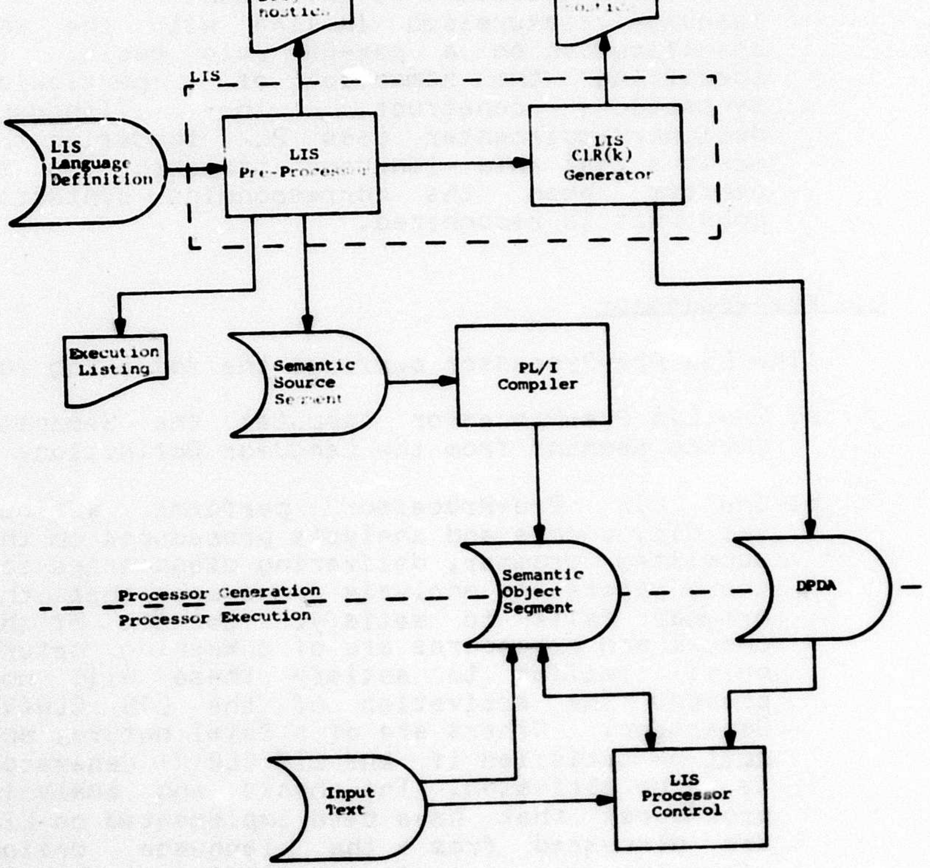
Processor Generation consists of the execution of the LIS Pre-Processor and the LIS CLR(k) Generator for purposes of computing the following functional results from the submitted LIS Language Definition:

a. The DPDAs which are used to "drive" LIS Processor Control in parsing legal Input Text of the language. Note that our use of the term, DPDA, is slightly different than as defined in Section II.B. Our present use of the term includes only the finite control. The input tape, output tape, and push down stack are incorporated into LIS Processor Control.

b. A PL/I procedure which represents the semantic interpretation to be associated with the language's syntactic constructs.

LIS Language Definition

A precise specification of the format of an LIS Language Definition may be found in Appendix A. For our present purposes, however, we may consider an LIS Language Definition to consist of:



Structure of the Language Implementation System

Figure II. 1.

a. A Backus Naur Form specification of the syntax of the language being defined.

b. A PL/I specification of the semantics of the language, expressed in-line with the BNF specification on a per-BNF rule basis. In specifying the semantics of a particular syntactic construct, the language designer/implementer uses PL/I to define the actions that his language processor is to perform when the corresponding syntactic construct is recognized.

<u>LIS Pre-Processor</u>

The LIS Pre-Processor performs the following functions:

a. The LIS Pre-Processor computes the Semantic Source Segment from the Language Definition.

b. The LIS Pre-Processor performs various validity checks and analysis procedures on the submitted grammar, delivering diagnostics for those checks and analysis procedures that the grammar fails to satisfy. Certain of the checks and procedures are of a warning nature only: failing to satisfy these will not prevent the activation of the LIS CLR(k) Generator. Others are of a fatal nature, and <u>must</u> be satisfied if the LIS CLR(k) Generator is to be activated. The checks and analysis procedures that have been implemented on LIS are discussed from the language design viewpoint in Appendix A, and from the LIS system design viewpoint in Appendix B.

<u>LIS CLR(k) Generator</u>

If the LIS Pre-Processor encounters no fatal errors in the LIS Language Definition, control is automatically passed to the LIS CLR(k) Generator. This phase of the system attempts to compute a CLR(k) parser for k less than or equal to a certain internally set value (currently set at 3). The

CLR(k) (Comprehensive Left to Right, looking ahead a maximum of k symbols) grammars constitute a large subset of the LR(k) grammars, which in turn possess the following characteristics:

a. The LR(k) condition generates exactly the deterministic context-free grammars.

b. The LR(k) grammars represent the largest class of grammars known to be parsable in linear time (proportional to the length of the input text) during a single left to right scan.

c. A grammar satisfying the LR(k) condition is unambiguous.

Intuitively, the LR(k) condition implies that the identity of a particular syntactic construct may be ascertained by looking indefinitely far to the left and at most k symbols to the right of the current position in the parse (symbols meaning characters or lexical constructs, depending on whether a lexical or primary grammar is being defined, respectively). This is an extremely comprehensive condition, and covers virtually all artificial languages that are likely to be of "practical" interest.

In attempting to compute the parsers, the LIS CLR(k) Generator delivers diagnostics for those areas of the language that do not satisfy the CLR(k) condition. These diagnostics include sufficient information on the language's local ambiguities to enable the language designer/implementer to modify the syntax of his language in order to make it CLR(k). Assuming that the grammar is CLR(k), the functional output of the LIS CLR(k) Generator is a segment containing one or two DPDAs, depending on whether a lexical parser, a primary parser, or both, are computed. The DPDAs, in combination with LIS Processor Control, constitute the parsers for the processor of the language being defined.

As a preview to the treatment in Chapter III, we note that the following algorithms are sequentially invoked by LIS in producing a DPDA from a given grammar:

1. Compute CFSM
   This algorithm is invoked to compute the Characteristic Finite State Machine of the submitted grammar.

2. Convert CFSM to DPDA
   This algorithm is invoked to convert the CFSM into a Deterministic Push Down Automaton. In converting each inadequate state of the CFSM, the algorithm invokes the CLR(k) look-ahead algorithm to associate a set of look-ahead transition strings with each of that state's generated DPDA states.

3. Optimize DPDA
   This algorithm is invoked to perform optimizations on the DPDA that will result in enhancements to the space and time efficiency of the resulting parser.

II.C.2 Processor Execution

A processor for the artificial language specified by the LIS Language Definition is synthesized by combining the DPDAs and the Semantic Object Segment with LIS Processor Control.

LIS Processor Control coordinates the overall language processing activity. In parsing the Input Text, it is "driven" by the DPDAs, and upon recognition of a particular syntactic construct, it activates the semantics associated with that construct. It is the responsibility of the activated semantics subsequently to return control to LIS Processor Control so that language processing may continue.

The semantics can access the Input Text directly, and the normal situation is for LIS Processor Control to coordinate these accesses by directing the semantics to specific text such as identifiers, keywords, symbol tables, etc. As indicated in Figure II.1, there is no explicit output from Processor Execution. It is therefore the responsibility of the semantics to manage its own output, as well as its alternate input files, temporary files, symbol tables, etc.

Chapter III

The Fundamental Algorithms of the Language Implementation System

III.A Introduction

In this chapter, we present the fundamental algorithms of the Language Implementation System. Being an algorithmic presentation, we will avoid much of the design and implementation detail. The reader interested in such detail is referred to Appendix B.

III.B An Example Grammar

Our approach in presenting the algorithms of LIS is to consider it our task to provide a processor (primary parser plus semantic controller) for a particular language, and to follow the language through the algorithms of Processor Generation that will compute from its primary grammar the required CLR(1) DPDA. We then present the CLR(1) parsing algorithm and thereby synthesize our language processor. The grammar for which we will produce a primary parser is as follows:

Preceding page blank

```
<lexical_non_terminal>  ::=
                        <identifier>  |
                        <integer>  !

<identifier>  ::=
                        <identifier> a->z  |
                        <identifier> 0->9  |
                        <identifier> _  |
                        a->z  !

<integer>  ::=
                        <integer> 0->9  |
                        0->9  !

<primary_non_terminal>  ::=
                        <assignment_statement>  !

<assignment_statement>  ::=
                        <identifier> = <expression> ;  !

<expression>  ::=
                        <expression> + <term>  |
                        <term>  !

<term>  ::=
                        <term> * <factor>  |
                        <factor>  !

<factor>  ::=
                        <identifier>  |
                        <integer>  !
```

This grammar defines a simple assignment statement language, the goal symbol of which is <primary_non_terminal>. It also defines <lexical_non_terminal>, and this definition serves to establish the "division of labor" between the constructs to be recognized by the primary parser, and those to be recognized by the lexical parser. The convention on LIS is that, with respect to the primary grammar, those non-terminals defined to be <lexical_non_terminal>s may be treated in the same way as terminal symbols. This point should be kept in mind during the subsequent discussions when we refer to terminal symbols and transitions on terminal symbols in the CFSM and DPDA.

At the end of this chapter, we demonstrate the CLR(1) parsing algorithm on sample text of our example grammar, and for this reason we will invoke LIS to compute both a lexical and a primary parser. However, our discussions on the fundamental algorithms will involve only the primary grammar.

III.C The Algorithm for Computing the Characteristic Finite State Machine

The algorithm that we use for computing the Characteristic Finite State Machine (CFSM) is that of Knuth-Earley. The CFSM is computed by iteratively generating unique configurations of the grammar and associating a CFSM state with each such configuration. A configuration may be thought of as a state of the grammar in which productions (alternatives) belonging to the state have exactly one of their symbols marked. The marked symbol of a production may be a non-terminal symbol, a terminal symbol or the production's #-symbol (application of alternative). Each marked production symbol of a configuration is associated with exactly one of the transitions of the corresponding CFSM state. It may be that several marked symbols correspond to the same transition.

In the subsequent discussion, we refer to the process of completing a configuration, by which we mean repeatedly scanning the particular configuration and adding to it new marked productions. One necessary condition for adding a marked production to a configuration is that there already exist a marked production whose marked symbol is a non-terminal. This being the case, we consider as candidates for marking and addition to the configuration, the leftmost symbols of all productions defining that non-terminal. An additional necessary condition for adding a marked production to a configuration is that the candidate (in the above sense) for addition must not already be a member of the configuration. Taken together, these conditions are both necessary and sufficient for adding marked productions to a configuration. The configuration is complete when a scan of the configuration does not result in the addition of new marked productions.

Since it is our task to produce a parser for our primary grammar, we know that we must be able to recognize an <assignment_statement>. We therefore initialize configuration-1 with the leftmost symbol of the production defining <primary_non_terminal> as follows (we indicate marked symbols with underlining):

```
     <primary_non_terminal> ::=
                                <assignment_statement> !
```

However, in the process of recognizing an <assignment_statement>, we must necessarily recognize an <identifier> (by definition of <assignment_statement>), so that we add to configuration-1 the following marked production:

```
     <assignment_statement> ::=
                                <identifier> = <expression> ; !
```

Since <identifier> is a <lexical_non_terminal>, we treat it as a terminal symbol, and therefore configuration-1 is complete as follows:

```
     Configuration-1
     <primary_non_terminal> ::=
                         <assignment_statement> !
     <assignment_statement> ::=
                         <identifier> = <expression> ; !
```

This configuration corresponds to our first CFSM state, which is automatically accessed when the parser is initiated. As we previously indicated, each marked production of a configuration corresponds to exactly one transition of the associated CFSM state. However, we do not yet know the destinations of the state transitions, so that state-1 is initially:

```
        State: 1  Accessed by:
                  <assignment_statement>     go to ?
                  <identifier>               go to ?
```

In order to determine the destination states of the transitions in state-1, it is necessary to go back to configuration-1 and consider the two configurations that would result by completing the initial configurations formed by advancing the marked symbols of configuration-1 one position to the right. Advancing the symbol marker corresponds to "reading" the symbol previously marked, which is exactly what we want. In the case of the first marked production of configuration-1, advancing the marker by one position results in the following initial configuration:

```
    <primary_non_terminal> ::=
                            <assignment_statement> ⊥
```

This newly marked symbol corresponds to the application of the production. Since the initial configuration has no marked non-terminals, the configuration is complete. During the process of CFSM computation, each configuration that we complete is considered temporary until it is determined whether the configuration has been previously generated. If the configuration *has* been previously generated, then the references that would be made to the temporary configuration will instead be made to the previously generated configuration, and the temporary configuration will be discarded. If the temporary configuration *has* *not* been previously generated, then it is retained. A scan of the configurations thus far generated (i.e. configuration-1) reveals no match with the temporary configuration, so that it is retained as configuration-2:

```
    Configuration-2
    <primary_non_terminal> ::=
                            <assignment_statement> ⊥
```

Thus, state-2 becomes the destination state of the first transition of state-1.

In determining the destination state of the second transition of state-1, we advance the marker on all marked productions of configuration-1 in which the marked symbol is <identifier>. Doing so produces the following initial configuration:

```
<assignment_statement> ::=
                        <identifier> ≡ <expression> ; !
```

Since the configuration does not contain a marked non-terminal, it is complete. A scan of the configurations thus far generated (configuration-1 and configuration-2) reveals no match with this temporary configuration, so that it is retained as configuration-3:

```
Configuration-3
<assignment_statement> ::=
                        <identifier> ≡ <expression> ; !
```

Thus, state-3 becomes the destination state of the second transition of state-1.

CFSM state-1 is now computed:

```
   State 1:  Accessed by:
             <assignment_statement>     go to 2
             <identifier>               go to 3
```

Turning now to state-2, we see that it is accessed by <assignment_statement>, and its corresponding configuration (configuration-2) consists of a single marked production. The marked symbol of the production is the application of alternative (4: 1). Since it is the application symbol that is marked, the marker cannot be further advanced within the production, and the corresponding state transition has no destination state.

CFSM state-2 is now computed:

```
   State: 2  Accessed by: <assignment_statement>
             Apply (4: 1)
```

Turning to state-3, we see that it is accessed by <identifier>, and that its corresponding configuration (configuration-3) consists of a single marked production. The symbol marked in configuration-3 is "=", so that state-3 is, initially:

```
   State: 3  Accessed by: <identifier>
             =            go to ?
```

In determining the destination state of the transition of state-3, we advance the marker in the marked production associated with the transition one position to the right. This yields the following initial configuration:

```
<assignment_statement> ::=
                         <identifier> = <expression> ; !
```

Completing this configuration results in a configuration matching none of the previously existing configurations, so that the new configuration is retained configuration-4:

```
Configuration-4
<assignment_statement> ::=
                       <identifier> = <expression> : !
<expression> ::=
                       <expression> + <term> ;
                       <term> !
<term> ::=
                       <term> * <factor> ;
                       <factor> !
<factor> ::=
                       <identifier> ;
                       <integer> !
```

CFSM state-3 is now computed:

```
   State: 3  Accessed by: <identifier>
             =            go to 4
```

Turning now to state-4, we see that it is accessed by "=", and that its corresponding configuration consists of 7 marked productions. In determining the destination state of the first transition of state-4, we advance the marker on all marked productions of configuration-4 in which the marked symbol is <expression>. This yields an initial configuration which has no marked non-terminals, and which is therefore complete. The configuration matches none of the previously existing configurations, and is therefore retained as configuration-5:

```
Configuration-5
<assignment_statement> ::=
                       <identifier> = <expression> _ !
<expression> ::=
                       <expression> + <term> ;
```

Thus, state-5 becomes the destination state of the first transition of state-4.

In determining the destination state for the second transition of state-4, we advance the marker on all productions in which the marked symbol is <term>. This yields an initial configuration which has no marked non-terminals, and which is therefore complete. The configuration matches none of the previously existing configurations, and is therefore retained as configuration-6:

Configuration-6

```
<term> ::=
                  <term> * <factor> |
                  <term> !
```

Thus, state-6 becomes the destination state of the second transition of state-4.

In determining the destination state of the third transition of state-4, we advance the marker on all marked productions of configuration-4 in which the marked symbol is <factor>. This yields an initial configuration which has no marked non-terminals, and which is therefore complete. The configuration matches none of the previously existing configurations, and is therefore retained as configuration-7:

Configuration-7

```
<term> ::=
                        <factor> i̲
```

Thus, state-7 becomes the destination state of the third transition of state-4.

In determining the destination state of the fourth transition of state-4, we advance the marker on all marked productions of configuration-4 in which the marked symbol is <identifier>. This yields an initial configuration which has no marked non-terminals, and which is therefore complete. The configuration matches none of the previously existing configurations, and is therefore retained as configuration-8:

Configuration-8

```
<factor> ::=
                        <identifier> i̲
```

In determining the destination state of the fifth transition of state-4, we move the marker on all marked productions of configuration-4 in which the marked symbol is <integer>. This yields an initial configuration which has no marked non-terminals, and which is therefore complete. The configuration matches none of the previously existing configurations, and is therefore retained as configuration-9:

Configuration-9
```
<factor> ::=
                   <integer> ⊥
```

Thus, state-9 becomes the destination state of the fifth transition of state-4.

CFSM state-4 is now computed:

```
   State: 4  Accessed by: =
             <expression>              go to 5
             <term>                    go to 6
             <factor>                  go to 7
             <identifier>              go to 8
             <integer>                 go to 9
```

We continue the state generation process until all CFSM states are computed. The process terminates when there are no more configurations to be processed into states. The completely computed CFSM is given at the end of this chapter. In addition to the information which we have been computing, the CFSM identifies, for each CFSM state, the type of state, the corresponding DPDA state, and in the case of an inadequate state, the DPDA apply state that corresponds to each apply transition of the state. For the apply transitions, the number of symbols in the alternative being applied is given, as is the non-terminal defined by the alternative.

The CFSM computation algorithm illustrated above may be formally specified as follows:

1. Set the number of configurations generated, n_configurations, equal to one.
   Set the number of CFSM states computed, n_cfsm_states, equal to one.
   Initialize configuration-1 with the leftmost symbol of all productions (alternatives) defining the grammar goal symbol (<primary_non_terminal> or <lexical_non_terminal>, as appropriate).
   Complete configuration-1.
   Go to step 2.

2. If n_cfsm_states is greater than n_configurations, then the CFSM is generated and the algorithm terminates.
   Otherwise, go to step 3.

3. Advance to the next marked symbol of the (n_cfsm_states)-th configuration that has not yet been converted into a state transition of the (n_cfsm_states)-th state.
   If none remain, then add one to n_cfsm_states and go to step 2.
   Otherwise, go to step 4.

4. The symbol detected in step 3 is the transition symbol for a new transition of the (n_cfsm_states)-th state. The new symbol may be a terminal symbol, a non-terminal symbol, or a #-symbol (application of production).
   Scan the rest of the (n_cfsm_states)-th configuration looking for all marked productions whose marked symbol is the same as the new transition symbol. These marked productions, as well as the one detected in step 3, should be identified so that they will not be considered in subsequent iterations of step 3 for this state.
   Go to step 5.

5. If the new transition symbol is a #-symbol, then the transition is determined, so go to step 3.
   Otherwise, go to step 6.

6. In order to determine the destination state of the new transition, initialize the (n_configurations + 1)-th configuration with the marked productions of the

```
   (n_cfsm_states)-th configuration which
   satisfied the conditions in steps 3 and 4.
   Advance the symbol markers of the marked
   productions one position to the right.
   Go to step 7.

7. Scan the n_configurations configurations
   looking for a match with the
   (n_configurations + 1)-th configuration. If a
   match is found, then:
      Set the destination state of the new
      transition to the number of the
      configuration matched.
      Destroy the (n_configurations + 1)-th
      configuration.
      Go to step 3.

   If a match is not found, then:
      Set the destination state of the new
      transition to n_configurations + 1.
      Retain the (n_configurations + 1)-th
      configuration.
      Add one to n_configurations.
      Go to step 3.
```

In the implementation of the CFSM computation algorithm, an efficient computational notation has been adopted for representing configurations. The notation involves associating a position in a bit vector with each symbol of each production (alternative) of the grammar, including #-symbols. Details of the representation of configurations and of the CFSM computation algorithm may be found in Appendix B.

III.D <u>The Algorithm for Converting the CFSM into a Deterministic Push Down Automaton</u>

Our approach in presenting the algorithm for converting the Characteristic Finite State Machine (CFSM) into a Deterministic Push Down Automaton (DPDA) is first to present the LR(O) CFSM parser. The inefficiencies and limitations inherent in the LR(O) parser will motivate first the stack algorithm and then the algorithm for converting the CFSM into a DPDA. We then present the conversion algorithm and apply it to the CFSM computed in the last section.

III.D.1 <u>The LR(O) CFSM Parser</u>

A grammar whose CFSM contains no inadequate states is LR(O). We now develop the CFSM parser for such a grammar and examine its limitations and inefficiencies. In the following discussion, we assume that we are to parse a sentence, T, of an LR(O) grammar, G. The parser will parse all canonical forms, CF, of G, and we initialize the canonical form with T, that is CF = T.

1. Initialize the CFSM in state-1, and apply it to the canonical form, CF.
   The parser will take transitions in the CFSM that correspond to the symbols of CF until an apply state, A, is reached.
   Go to step 2.

2. The production to be applied in state-A is A->w.
   Output A and replace the w just read in the CF

with A. This is the new CF.
Go to step 3.

3. If CF = S (goal symbol of grammar), then the parse is complete.
Otherwise, go to step 1.

III.D.2 The Stack Algorithm

The LR(0) CFSM parser is grossly inefficient. Our particular objection is it's rescan of previously scanned portions of canonical forms. Since the parser is deterministic, this rescanning is simply wasted processing. Our solution is to save information on the parse in a push down stack so as to remove the requirement for rescanning. This will be our stack algorithm.

Consider a single iteration of the parser. The canonical form for the iteration is initially CF' = rwb, and after scanning rw, the parser ends up at an apply state in which the production being applied is A->w. Application of the production results in the new canonical form, CF = rAb. However, applying the parser to CF will take it through the same set of states in recognizing r, (i.e. the CFSM is deterministic), so that, had it remembered the state entered just after reading r, it could start in that state on the canonical form, Ab, and get the same result as if it had started in state-1 on rAb. It is clear, then, that if we keep a push down stack of all states entered by the CFSM, we can maintain a history of the parse that is relevant at production application time. Then, when the parser scans rw and enters the state in which a->w is applied, it simply pops |w| states off the stack (where |w| means the number of symbols in w) and resumes the parse of Ab in the state that is on the top of the stack. This process of popping and resuming the parse in the top state of the stack is called look-back. As before, the parser stops when CF = S.

The stack algorithm is equivalent to the LR(0) CFSM parser in terms of the resulting parse, and it is far more efficient. Thus, the primary motivation for development of the stack algorithm is the increase in parse time efficiency that may be obtained. Note, however, that the stack algorithm has only one interaction with the symbols of the input text, the initial scan up to the application of the associated production. Thereafter, all manipulations are in terms of the states on the stack and the non-terminal transitions from the look-back states. Thus, were it not for the need for the non-terminal transitions from look-back states, all non-terminal transitions of the CFSM could be ignored. As will be seen, the conversion algorithm takes care of this by modifying the look-back states in such a way that all non-terminal transitions of the CFSM are deleted.

Although the development of the stack algorithm is important for reasons of efficiency, the primary motivation for the development of the conversion algorithm about to be presented is that very few grammars of "practical" value are LR(0). We must therefore introduce look-ahead to resolve the inadequate CFSM states.

III.D.3 <u>The Conversion Algorithm</u>

The conversion algorithm will be applied to the CFSM computed in Section III.B, and will produce the <u>initial</u> (non-optimized) DPDA given at the end of the chapter. The basic steps of the conversion algorithm are as follows:

1. Convert each CFSM state containing read transitions into a separate DPDA read state, eliminating all transitions on non-terminal symbols.
2. Convert each CFSM apply transition into a separate DPDA apply state with look-back transitions.
3. Convert the cfsm inadequate states into DPDA look-ahead states.

<u>Conversion to Read States</u>

The conversion of CFSM read states into DPDA read states is straightforward. Simply go through the CFSM and establish a DPDA read state for each CFSM read state. Then, for each CFSM read state, move into the corresponding DPDA read states, only those transitions whose transition symbol is a terminal symbol.

An inadequate CFSM state containing read transitions also generates a DPDA read state. As with the conversion of CFSM read states, we go through the inadequate CFSM state and move only those transitions whose transition symbol is a terminal symbol.

Applying the conversion processes just described to the CFSM computed in Section III.C results in the DPDA read states of the initial DPDA given at the end of the chapter. The parenthesized state numbers in the DPDA refer to states of the CFSM.

Conversion to Apply States

Each apply transition of the CFSM generates a separate DPDA apply state. As indicated in the discussion of the stack algorithm, the apply state can be provided with look-back information which will enable the parse to resume in the state entered just prior to beginning the scan of the symbols on the right side of the applied production. In the case of the stack algorithm, once it was determined in which state to resume the parse, the first transition taken was on the non-terminal produced in the apply state. Since this is known a priori, the look-back information can be extended so as to include the destination state of this non-terminal transition. The extended look-back states are referred to as the look-back transitions of the DPDA apply state, or alternatively, as the top transitions, or the apply transitions of the state. Incorporating the look-back transitions into the parser guarantees that no transition will ever be taken on a non-terminal, so that no non-terminal transitions need appear in the DPDA.

We now give a procedure for computing the look-back transitions of a particular DPDA apply state. The look-back states of an apply transition of the CFSM are those CFSM states from which originate a transition path over the symbols of the applied production, the path terminating in the CFSM state to which the apply transition belongs. During the computation of the CFSM (specifically, during the process of completing a configuration), it is a simple matter to keep a list of the states from which originate transition paths corresponding to the right side of the grammar's productions. Following the completion of the CFSM computation, we go through this list, and, for each entry, go to the indicated CFSM state, C, and follow a transition path corresponding to the right side of the indicated production, P. This path will terminate in a CFSM state, A, containing an application of the production, P. The DPDA read state corresponding to CFSM state C will be a look-back state of the DPDA apply state corresponding to the application of P in A. The destination state of the apply transition is simply the DPDA state corresponding to the destination state of the transition in CFSM state C which has as its transition symbol, the non-terminal defined in production P.

The number of symbols popped in a particular DPDA apply state will be one less than the number of symbols in the production being applied in that state. This is because only the read states of the DPDA push their state numbers onto the state stack.

Applying the conversion processes just described to the CFSM computed in Section III.C results in the DPDA apply states indicated in the _initial_ DPDA at the end of the chapter. Again, the parenthesized state numbers refer to CFSM states.

Conversion to Look-Ahead States

An inadequate CFSM state is inadequate in the sense that the CFSM parsing algorithm cannot remain deterministic when encountering such a state. This is because information does not exist within the state which is capable of indicating the transition to be taken on entering that state. To remedy this, each inadequate CFSM state converts into a DPDA look-ahead state. Also, each inadequate state generates a separate DPDA apply state for each of its apply transitions, and a single DPDA read state for its set of read transitions (if any). In effect, the look-ahead state contains the potential symbol strings (look-ahead strings) that could be encountered on pursuing transitions through the generated states. By pre-computing and saving this look-ahead information, and associating each look-ahead string with the appropriate generated state, we maintain the determinism of our parsing algorithm. This is because we can compare the look-ahead symbol strings with the symbols actually occurring ahead in the input text, and take the transition on which we get a match. If we do not get a match on any of the strings, then the input text contains a syntax error.

The actual computation of the look-ahead symbol strings is done by the look-ahead algorithm, which we are about to present. By applying the conversion process and computing the look-ahead symbols with the look-ahead algorithm, we convert the inadequate states of the CFSM computed in Section III.C into the DPDA look-ahead states in the _initial_ DPDA at the end of the chapter. The destination state numbers indicated in these DPDA look-ahead transitions refer to DPDA states.

The Look-Ahead Algorithm

The look-ahead algorithm is invoked by the CFSM to DPDA conversion algorithm for purposes of resolving inadequate CFSM states. The look-ahead algorithm takes as its input, the CFSM, the number of the inadequate state to be resolved, and the length, k, of the look-ahead strings by which resolution is to be attempted. The output of the algorithm is a set of look-ahead transitions, each transition associating a k-symbol look-ahead string with a destination state. Recall that each inadequate CFSM state generates a separate DPDA apply state for each of its apply transitions, and generates a single DPDA read state for its set of read transitions. These generated states are the destination states of the look-ahead transitions.

The look-ahead algorithm leaves to the CFSM to DPDA conversion algorithm the task of determining whether the inadequacy has been resolved for the particular state and value of k. The condition for resolution is simply that no look-ahead string may occur in two or more look-ahead transitions having different destination states. If the attempt at resolution is successful, the conversion algorithm saves the look-ahead state and its transitions and proceeds to attempt resolution of the next inadequate CFSM state. If resolution is not successful, the value of k is increased by one and the look-ahead algorithm is once again invoked. Resolution is attempted for values of k=1,2, and 3. If resolution is unsuccessful after three symbol look-ahead, the attempt at state resolution is abandoned, error diagnostics are issued, and the inadequate CFSM state is designated unresolved.

The essence of look-ahead is that, for each destination state (in the above sense), a computation is performed to determine the set of all possible transition strings of k terminal symbols (look-ahead strings) that can be encountered, given entry into the particular destination state. These look-ahead string/destination state pairs constitute the look-ahead transitions of the look-ahead state for the particular value of k.

The algorithm operates on the information local to the inadequate state in the CFSM. In computing the transition symbol strings, the algorithm may encounter CFSM read states, CFSM apply states, and inadequate CFSM states. Let us assume that on entering one of these states, the look-ahead string under construction is S, and that S has been built up to a length of l symbols (l=0,1,2). The algorithm is thus looking to add k-l symbols to S, and its action in each case is as follows:

CFSM read state

Assume that the state has n transitions on terminal symbols. Then the look-ahead string, S, is duplicated n times, and for each of the n transitions, the transition symbol is concatenated to a copy of S, resulting in a new transition symbol string, S′, of length l+1. Now, if l+1=k, then S′ is complete. However, if l+1<k, then the look-ahead algorithm is reapplied at the destination state of the transition. On reapplication, the look-ahead string is S′ and the algorithm is looking to add k-l-1 symbols to S′.

CFSM apply state

The CFSM apply state has an associated DPDA apply state for which has been computed a set of look-back transitions. The look-ahead algorithm is reapplied, with S, to the destination state of each look-back transition.

CFSM Inadequate State

The look-ahead algorithm as applied to a CFSM inadequate state may be viewed as its repeated application to as many CFSM read states and CFSM apply states as necessary to represent all transitions of the inadequate state, each application being performed with S.

With respect to the reapplication of the look-ahead algorithm, we place the restriction that the algorithm not be reapplied to a CFSM state to which it was previously applied if the read transitions constituting the look-ahead string up to the proposed reapplication are identical to those up to the previous application. This condition includes the null look-ahead string and prevents the algorithm from entering infinite cycles.

The look-ahead algorithm is conceptually quite simple. The complexity of its implementation comes from doing the accounting associated with keeping track of each look-ahead string and the states that have been entered at each level on its behalf. As an accounting aid, we have found the concept of look-ahead contours to be useful. A look-ahead contour represents a set of look-ahead symbols for each symbol level (value of k). Each contour contains a number of states, and transitions between states are within the same contour (reapplication), or from lower contours to higher contours on single contour transitions (read transitions). A dashed line transition represents the reapplication of the algorithm based on the occurrence of an apply transition. A solid line transition represents the application of the look-ahead algorithm to a read transition, the transition symbol being the label of the transition. The look-ahead strings are defined by the labels on the read transitions of continuous directed transition sequences.

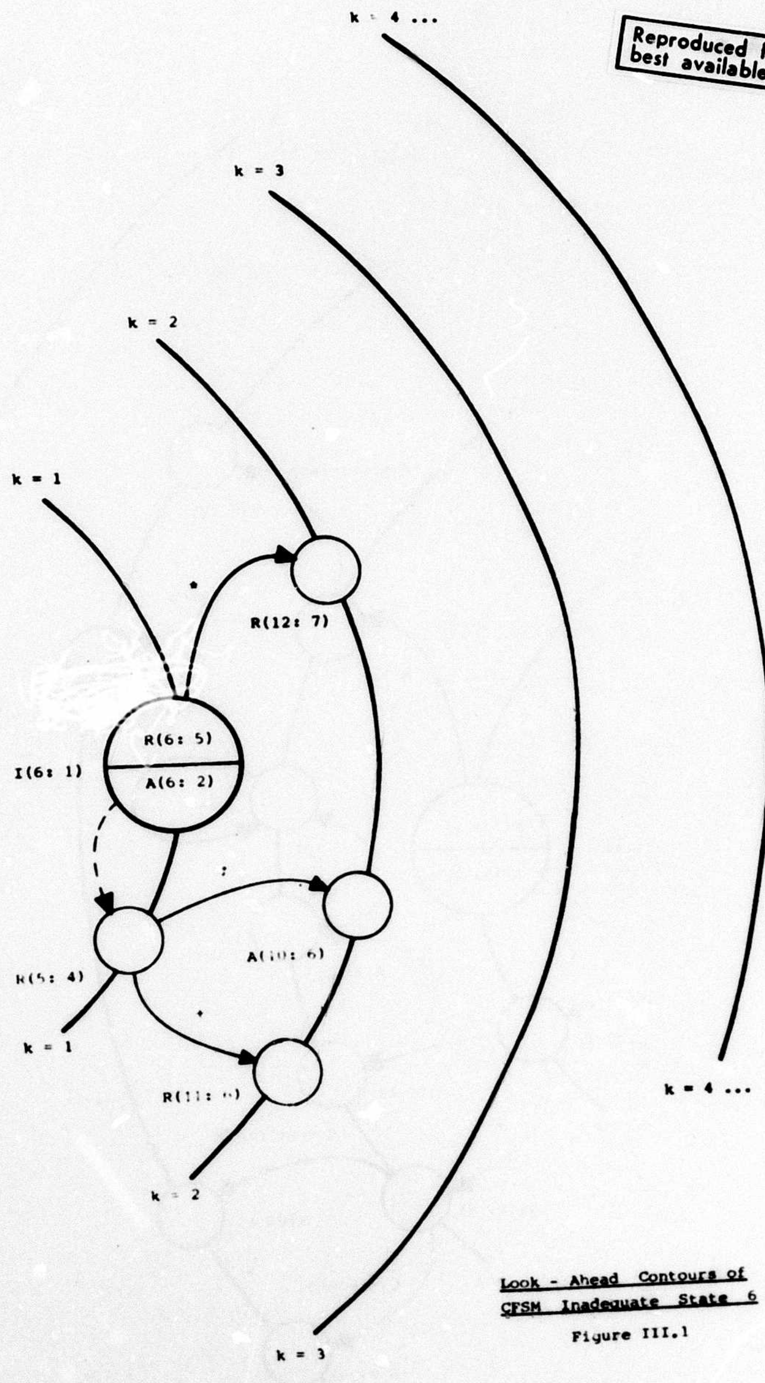
In Figures III.1 and III.2, we use the look-ahead contours in computing the look-ahead transitions of the two look-ahead states of our example grammar. In the process of contour generation, we label states in the following way:

S(C: D)

In this notation, S may be R (CFSM Read state), A (CFSM Apply state), or I (CFSM Inadequate state). C is the number of a CFSM state, and D is the number of a DPDA state. We also use the notation to represent the DPDA read states and the DPDA apply states generated from inadequate CFSM states.

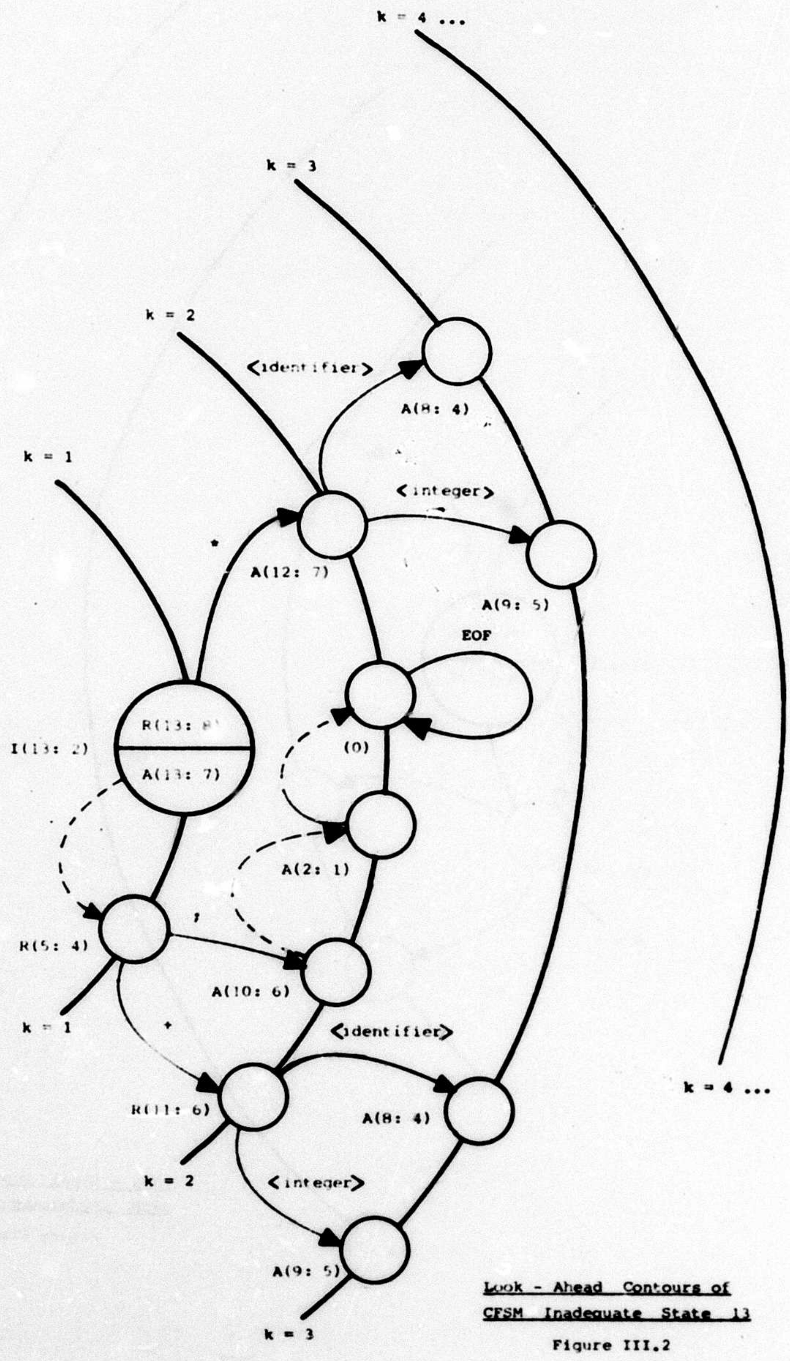
Our example grammar is CLR(1), and its inadequacies can thus be resolved by our algorithm with one symbol look-ahead. In Figure III.1, we simply perform the one symbol look-ahead for the first inadequate state. In Figure III.2, we generate two contours for the second inadequate state, so as to give a more thorough example of the look-ahead algorithm in action.

Reproduced from best available copy.



Look - Ahead Contours of CFSM Inadequate State 6

Figure III.1



<u>Look - Ahead Contours of</u>
<u>CFSM Inadequate State 13</u>
Figure III.2

III.E <u>The Algorithm for Optimizing the DPDA</u>

The algorithms that we discuss in this section have to do with the optimization of the <u>contents</u> of the DPDA without regard for its representation in a particular computing environment. That is, transformations are performed on the DPDA that remove superfluous and redundant information so that the resulting DPDA is more efficient than its predecessor. The optimizations that have been implemented by no means exhaust the potential for DPDA content optimization. Other optimizations, such as transition sorting according to empirical measures of frequency of transition occurrence for a particular language, and detection and deletion of apply states that have no associated semantics and that do not modify the DPDA state stack, are but a few of the optimizations that could have significant impact on parser space and time efficiency.

The representation of the DPDA read states is such that the information regarding the states themselves is stored separately from the information on the state's transitions. This being the case, we can optimize the read transitions by deleting duplicate transition sequences that may arise from different read states.

There are two fundamental optimizations that are performed on the DPDA apply states. First, for each apply state, we determine the most popular look-back transition destination state, and designate that the default destination state. Then all look-back transitions of the state whose destination state is the default destination state are deleted from the list of look-back transitions. The default destination state is then appended to the list, it being the convention that, during parsing, should the top of the state stack (after being popped) fail to match any of the look-back states in the list for the current apply state, then the transition to the default destination state is automatically taken. Since the CLR(k) parser is deterministic, we are guaranteed not to introduce any errors by performing this optimization.

The second optimization that we apply to the DPDA apply states is analogous to the optimization applied to the DPDA read states, and we thus remove redundant information.

The number of optimizations performed on the DPDA look-ahead states is one or two, depending on whether the grammar in question is lexical or primary, respectively. In either case, duplicate look-ahead transitions for a given look-ahead state are deleted. In the case of a parser computed from a primary grammar, an additional optimization is performed on the look-ahead states which is analogous to the first of the optimizations applied to the DPDA apply states. Thus, for each look-ahead state, we determine the most popular look-ahead transition destination state, and designate that the default destination state. Anv look-ahead transition whose destination state matches the default destination state is deleted from the list of look-ahead transitions for the state in question, and the default destination state is appended to the end of the list. The parsing interpretation of the default look-ahead transition is analogous to the parsing interpretation of the default apply transition. However, in the present case, the detection of an erroneous symbol in the input stream will be delayed until a subsequent read state, whereas were the default destination optimization not performed, such an error would be detected in the look-ahead state.

The above optimizations have been applied to the initial DPDA to produce the final DPDA given at the end of the chapter. Note that in this final DPDA we have replaced all parenthesized CFSM state numbers with the corresponding DPDA state numbers.

In addition to the DPDA content optimizations, a significant improvement in parser space and time efficiency may be realized by "fine tuning" the DPDA to the particular computing system on which the parser is to be executed.

This may involve the packing of integer fields into bit strings, the hashing of the various state transitions, and even the generation of an assembly code representation of the DPDA and its control procedure. We hesitate to make generalizations about the type of representation optimizations that can be performed, since the range of possibilities is limited only by one's imagination and the space-time tradeoffs inherent in the computing environment under consideration. Readers interested in the optimizations that we have performed in the representation of the DPDAs on Multics are referred to Appendix B.

III.F The CLR(1) Parsing Algorithm

In this section, we present the basic CLR(1) parsing algorithm that "drives" the language processors produced by LIS. We present the CLR(1) algorithm because of its simplicity and because it has been our experience that one symbol look-ahead is sufficient for most applications. Extension of the algorithm to CLR(k) for k>1 is straightforward. Readers wishing more detail on the CLR(1) algorithm are referred to Section B.2.11 of Appendix B. Examples of the execution of the parser on text of our example grammar are given at the end of the chapter.

The CLR(1) parser is an extension of the stack algorithm presented in Section III.D. The stack algorithm is extended to incorporate the look-back transitions of apply states and the look-ahead transitions of look-ahead states. In the following discussion, we assume that we are driving a primary parser, and that a lexical parser exists that provides lexical constructs on demand (of course, our algorithm may also be adapted to lexical parsing, as indicated in Appendix B). We fetch a construct by envoking the procedure FETCH_CONSTRUCT, the fetched construct being placed in CONSTRUCT. Our discussion also makes reference to two stacks, a DPDA state stack and a text reference stack. The DPDA state stack is the stack of DPDA read states that is maintained so as to implement the look-back transitions of apply states. The text reference stack contains the lexical constructs as recognized by the lexical parser, and represents the primary interface of the language semantics with the input text. The use of this stack is explained in Appendix A.

The CLR(1) parsing Algorithm:

```
1. Initialization
   Perform the initialization semantics specified
   in the language definition.
   FETCH_CONSTRUCT, have_construct = "yes".
   Clear DPDA state stack, Clear text reference
   stack.
   Go to Step 3.

2. Next State
   If STATE is a DPDA read state, go to step 3.
   If STATE is a DPDA apply state, go to step 4.
   IF STATE is a DPDA look-ahead state, go to
   step 5.

3. DPDA Read State
   If have_construct = "no", FETCH_CONSTRUCT.
   have_construct = "no".
   Push STATE onto DPDA state stack.
   Push CONSTRUCT onto text reference stack.
   Scan transitions of STATE looking for match
   with CONSTRUCT.
   If match not found, then a syntax error has
   been detected, so exit to error reporting and
   recovery procedure (Section IV.C).
   Otherwise, set STATE to the transition
   destination state of the matching transition.
   Go to step 2.

4. DPDA Apply State
   If semantics is associated with the BNF rule
   to which the production to be applied belongs,
   activate the semantics.
   Pop the DPDA state stack as many times as
   indicated for STATE.
```

If the production being applied defines <primary_non_terminal>, then processing is complete, so exit.
Scan look-back transitions for STATE looking for a match with the top of the DPDA state stack.
If a match is found, set STATE to the destination state of the matching look-back transition.
If a match is not found, set STATE to the default destination state for STATE.
Go to step 2.

5. <u>DPDA Look-Ahead State</u>
If have_construct = "no", FETCH_CONSTRUCT.
have_construct = "yes"
Scan look-ahead transitions for STATE, looking for a match with CONSTRUCT.
If a match is found, set STATE to the destination state of the matching look-ahead transition.
If a match is not found, set STATE to the default destination state of STATE.
Go to step 2.

```
         The  <primary_non_terminal>  CFSM  For:  >udd>LIS>Altmanv>LIS_DOCUMENTATION>example.lis


State: 1   Accessed by:                        read      OPDA State: 1
           <assignment_statement>              go to 2
           <identifier>             go to 3

State: 2   Accessed by: <assignment_statement>                       apply      OPDA State: 1
           Apply (1: 1)             OPDA Apply State: 1              Pop 1      -> <primary_non_terminal>

State: 3   Accessed by: <identifier>                       read      OPDA State: 2
           -           go to 4

State: 4   Accessed by: =                      read      OPDA State: 3
           <expression>             go to 5
           <term>      go to 6
           <factor>    go to 7
           <identifier>             go to 8
           <integer>   go to 9

State: 5   Accessed by: <expression>                       read      OPDA State: 4
           ;           go to 10
           +           go to 11

State: 6   Accessed by: <term>              ** INADEQUATE **      OPDA State: 1
           *           go to 12
           Apply (3: 2)        OPDA Apply State: 2                   Pop 1      -> <expression>

State: 7   Accessed by: <factor>                           apply     OPDA State: 3
           Apply (4: 2)        OPDA Apply State: 3                   Pop 1      -> <term>

State: 8   Accessed by: <identifier>                       apply     OPDA State: 4
           Apply (5: 1)        OPDA Apply State: 4                   Pop 1      -> <factor>

State: 9   Accessed by: <integer>                          apply     OPDA State: 5
           Apply (5: 2)        OPDA Apply State: 5                   Pop 1      -> <factor>

State: 10  Accessed by: ;                  apply     OPDA State: 6
           Apply (2: 1)        OPDA Apply State: 6                   Pop 4      -> <assignment_statement>

State: 11  Accessed by: +                  read      OPDA State: 6
           <term>      go to 13
           <factor>    go to 7
           <identifier>             go to 8
           <integer>   go to 9

State: 12  Accessed by: *                  read      OPDA State: 7
           <factor>    go to 14
           <identifier>             go to 8
           <integer>   go to 9

State: 13  Accessed by: <term>              ** INADEQUATE **      OPDA State: 2
           *           go to 12
           Apply (3: 1)        OPDA Apply State: 7                   Pop 3      -> <expression>

State: 14  Accessed by: <factor>                           apply     OPDA State: 8
           Apply (4: 1)        OPDA Apply State: 8                   Pop 3      -> <term>
```

```
          The  Initial  <primary_non_terminal>  DPDA  For:  >udd>LIS>Alteenv>LIS_DOCUMENTATION>example.lis



          The <primary_non_terminal> Read States:


Read State: 1
          read <identifier>   go to read state (3)


Read State: 2
          read =    go to read state (4)


Read State: 3
          read <identifier>   go to apply state (8)
          read <integer>      go to apply state (9)


Read State: 4
          read ;    go to apply state (10)
          read +    go to read state (11)


Read State: 5
          read *    go to read state (12)


Read State: 6
          read <identifier>   go to apply state (8)
          read <integer>      go to apply state (9)


Read State: 7
          read <identifier>   go to apply state (8)
          read <integer>      go to apply state (9)


Read State: 8
          read *    go to read state (12)
```

```
        The  <primary_non_te-minal>  Apply  States:


Apply State: 1      apply production: (1: 1)          pop 0    -> <priaary_non_terminal>
        top (1)     go to read state (0)

Apply State: 2      apply production: (3: 2)          pop 0    -> <expression>
        top (4)     go to raad state (5)

Apply State: 3      apply production: (4: 2)          pop 0    -> <tera>
        top (4)     go to look-ahead state (6)
        top (11)    go to look-ahaad state (13)

Apply State: 4      apply production: (5: 1)          pop 0    -> <factor>
        top (4)     go to apply state (7)
        top (11)    go to apply state (7)
        top (12)    go to apply state (14)

Apply State: 5      apply production: (5: 2)          pop 0    -> <factor>
        top (4)     go to apply state (7)
        top (11)    go to apply state (7)
        top (12)    go to apply stata (14)

Apply Statal 6      apply production: (2: 1)          pop 3    -> <assignment_statemant>
        top (1)     go to apply state (2)

Apply State: 7      apply production: (3: 1)          pop 2    -> <expression>
        top (4)     go to raad state (5)

Apply Statal 8      apply production: (4: 1)          pop 2    -> <tera>
        top (4)     go to look-ahead stata (6)
        top (11)    go to look-ahead stata (13)


        The <priaary_non_terminal> Look - Ahead States:


Look - Ahaad State: 1
        sea *       go to read stata 5
        see ;       go to apply state 2
        see +       go to apply state 2

Look - Ahead State: 2
        see *       go to raad state 8
        see ;       go to apply state 7
        sea +       go to apply stata 7
```

```
The  <primary_non_terminal>  DPDA  For:  >udd>LIS>Alteenv>LIS_DOCUMENTATION>example.lis



          The <primary_non_terminal> Read States:

                                   8 States, 7 Transitions, Maximum Transitions per State = 2


Read State: 1
          read <identifier>   go to read state 2

Read State: 2
          read =    go to read state 3

Read State: 3
          read <identifier>   go to apply state 4
          read <integer>      go to apply state 5

Read State: 4
          read ;    go to apply state 6
          read +    go to read state 6

Read State: 5
          read *    go to read state 7

Read State: 6
          read <identifier>   go to apply state 4
          read <integer>      go to apply state 5

Read State: 7
          read <identifier>   go to apply state 4
          read <integer>      go to apply state 5

Read State: 8
          read *    go to read state 7
```

```
        The  <primary_non_terminal>  Apply  States:

                                        8 States, 7 Transitions, Maximum Transitions per State = ?

                                              4
Apply State: 1      apply production: (~~1~~: 1)          pop 0    -> <primary_non_terminal>
        PARSE COMPLETED.

                                              6
Apply State: 2      apply production: ([illegible]: 2)          pop 0    -> <expression>
                    go to read state 4

                                              7
Apply State: 3      apply production: ([illegible]: 2)          pop 0    -> <term>
        top 3       go to look-ahead state 1
                    go to look-ahead state 2

                                              8
Apply State: 4      apply production: ([illegible]: 1)          pop 0    -> <factor>
        top 7       go to apply state 8
                    go to apply state 3

                                              8
Apply State: 5      apply production: ([illegible]: 2)          pop 0    -> <factor>
        top 7       go to apply state 8
                    go to apply state 3

                                              5
Apply State: 6      apply production: ([illegible]: 1)          pop 3    -> <assignment_statement>
                    go to apply state 1

                                              6
Apply State: 7      apply production: ([illegible]: 1)          pop 2    -> <expression>
                    go to read state 4

                                              7
Apply State: 8      apply production: ([illegible]: 1)          pop 2    -> <term>
        top 3       go to look-ahead state 1
                    go to look-ahead state 2



        The <primary_non_terminal> Look - Ahead States:

                                        2 States, 4 Transitions, Maximum Transitions per State = 2


Look - Ahead State: 1
        see *       go to read state 5
                    go to apply state 2

Look - Ahead State: 2
        see *       go to read state 8
                    go to apply state 7
```

```
example
example: arguments missing.
Arguments:
1:     <example_input_text_segment_name>
2-4:   <options>
          "p"        Print parse of example program.
          "ns"       Do not perform any translation
                     semantics.

print el.example 1
a = b + c;


example el ns p
The LIS CLR(k) Parse of el.example:
                                                                Stack (<- top)
Line       Action                                               1¦
1          read a     go to read state 2                        1¦2¦
1          read =     go to read state 3                        1¦2¦3¦
1          read b     go to apply state 4                       1¦2¦3¦
1          apply (8: 1)   pop 0, go to apply state 3            1¦2¦3¦
1          apply (7: 2)   pop 0, go to look-ahead state 1       1¦2¦3¦
1          see +      go to apply state 2                       1¦2¦3¦
1          apply (6: 2)   pop 0, go to read state 4             1¦2¦3¦4¦
1          read +     go to read state 6                        1¦2¦3¦4¦6¦
1          read c     go to apply state 4                       1¦2¦3¦4¦6¦
1          apply (8: 1)   pop 0, go to apply state 3            1¦2¦3¦4¦6¦
1          apply (7: 2)   pop 0, go to look-ahead state 2       1¦2¦3¦4¦6¦
1          see ;      go to apply state 7                       1¦2¦3¦
1          apply (6: 1)   pop 2, go to read state 4             1¦2¦3¦4¦
1          read ;     go to apply state 6                       1¦
1          apply (5: 1)   pop 3, go to apply state 1            1¦
1          apply (4: 1)   pop 0, go to read state 0

Parse Completed
```

```
print e2.example 1
left = operand_1 + 2*operand_2 +
operand_3 +
8*operand_4*operand_5 * 54321
+ operand_6 + operand_7;


example e2 ns p
The LIS CLR(k) Parse of e2.example:

Line     Action                                            Stack (<- top)
1        read left go to read state 2                      1:
1        read =    go to read state 3                      1:2:
1        read operand_1      go to apply state 4           1:2:3:
1        apply (8: 1)   pop 0, go to apply state 3         1:2:3:
1        apply (7: 2)   pop 0, go to look-ahead state 1    1:2:3:
1        see +     go to apply state 2                     1:2:3:
1        apply (6: 2)   pop 0, go to read state 4          1:2:3:
1        read +    go to read state 6                      1:2:3:4:
1        read 2    go to apply state 5                     1:2:3:4:6:
1        apply (8: 2)   pop 0, go to apply state 3         1:2:3:4:6:
1        apply (7: 2)   pop 0, go to look-ahead state 2    1:2:3:4:6:
1        see *     go to read state 8                      1:2:3:4:6:
1        read *    go to read state 7                      1:2:3:4:6:8:
1        read operand_2      go to apply state 4           1:2:3:4:6:8:7:
1        apply (8: 1)   pop 0, go to apply state 8         1:2:3:4:6:8:7:
1        apply (7: 1)   pop 2, go to look-ahead state 2    1:2:3:4:6:
1        see +     go to apply state 7                     1:2:3:4:6:
1        apply (6: 1)   pop 2, go to read state 4          1:2:3:
1        read +    go to read state 6                      1:2:3:4:
2        read operand_3      go to apply state 4           1:2:3:4:6:
2        apply (8: 1)   pop 0, go to apply state 3         1:2:3:4:6:
2        apply (7: 2)   pop 0, go to look-ahead state 2    1:2:3:4:6:
2        see +     go to apply state 7                     1:2:3:4:6:
2        apply (6: 1)   pop 2, go to read state 4          1:2:3:
```

```
2     read +     go to read state 6                                1:2:3:4:
3     read 8     go to apply state 5                               1:2:3:4:6:
3     apply (8: 2)   pop 0, go to apply state 3                    1:2:3:4:6:
3     apply (7: 2)   pop 0, go to look-ahead state 2               1:2:3:4:6:
3     see *      go to read state 8                                1:2:3:4:6:
3     read *     go to read state 7                                1:2:3:4:6:8:
3     read operand_4       go to apply state 4                     1:2:3:4:6:8:7:
3     apply (8: 1)   pop 0, go to apply state 8                    1:2:3:4:6:8:7:
3     apply (7: 1)   pop 2, go to look-ahead state 2               1:2:3:4:6:
3     see *      go to read state 8                                1:2:3:4:6:
3     read *     go to read state 7                                1:2:3:4:6:8:
3     read operand_5       go to apply state 4                     1:2:3:4:6:8:7:
3     apply (8: 1)   pop 0, go to apply state 8                    1:2:3:4:6:8:7:
3     apply (7: 1)   pop 2, go to look-ahead state 2               1:2:3:4:6:
3     see *      go to read state 8                                1:2:3:4:6:
3     read *     go to read state 7                                1:2:3:4:6:8:
3     read 54321           go to apply state 5                     1:2:3:4:6:8:7:
3     apply (8: 2)   pop 0, go to apply state 8                    1:2:3:4:6:8:7:
3     apply (7: 1)   pop 2, go to look-ahead state 2               1:2:3:4:6:
4     see +      go to apply state 7                               1:2:3:4:6:
4     apply (6: 1)   pop 2, go to read state 4                     1:2:3:
4     read +     go to read state 6                                1:2:3:4:
4     read operand_6       go to apply state 4                     1:2:3:4:6:
4     apply (8: 1)   pop 0, go to apply state 3                    1:2:3:4:6:
4     apply (7: 2)   pop 0, go to look-ahead state 2               1:2:3:4:6:
4     see +      go to apply state 7                               1:2:3:4:6:
4     apply (6: 1)   pop 2, go to read state 4                     1:2:3:
4     read +     go to read state 6                                1:2:3:4:
4     read operand_7       go to apply state 4                     1:2:3:4:6:
4     apply (8: 1)   pop 0, go to apply state 3                    1:2:3:4:6:
4     apply (7: 2)   pop 0, go to look-ahead state 2               1:2:3:4:6:
4     see ;      go to apply state 7                               1:2:3:4:6:
4     apply (6: 1)   pop 2, go to read state 4                     1:2:3:
4     read ;     go to apply state 6                               1:2:3:4:
4     apply (5: 1)   pop 3, go to apply state 1                    1:
```

```
4         apply (4: 1)   pop 0, go to read state 0          1:

Parse Completed
```

Chapter IV

Conclusions

IV.A Introduction

In this chapter, we consider a number of issues. In Section IV.B, we present empirical evidence supporting our previous claims regarding the efficiency of CLR(k) parsing. In Section IV.C. we discuss the design of an important enhancement to the Language Implementation System, namely syntax directed error detection, reporting, and recovery procedures. In Section IV.D, we briefly discuss the hierarchy of LR(k) systems and indicate the position of the CLR(k) grammars within this hierarchy. In Section IV.E, we consider some of the significant language developments in which LIS has been utilized. In Section IV.F, we discuss areas of future research and development that may be expected to have significant impact on language implementation system technology.

IV.B The Efficiency of CLR(k) Parsers

Efficiency of parsing strategies is typically analyzed along two dimensions, space and time. Space efficiency refers to the space requirements of the parsing strategy, whereas time efficiency refers to parsing speed. In this section, we investigate the space and time efficiencies of the CLR(k) parsers produced by LIS. Our presentation will be in three parts. First, we report on the empirical efficiency comparisons made at the University of Toronto between the parsers produced by their LALR(k) generator and parsers produced by popular precedence methods. Then we report on the efficiency characteristics of the primary parsers produced by LIS. Finally, we indicate the way in which we have adapted our CLR(k) strategy to the production of efficient lexical parsers.

IV.B.1 The Toronto LALR(k)/Precedence Comparisons

The most meaningful empirical investigations into the efficiency of parsing strategies are those which compare alternative strategies across a common base of languages in a common computing environment. Unfortunately, this type of comparison was not possible in the case of LIS, since no other automatic strategies exist on Multics. However, the Computer Systems Research Group at the University of Toronto performed exactly these types of comparisons on an IBM System/360 (Model 44), The comparisons were made among the LALR(k) parsers (see Section IV.D) produced by their system

(La 71, Hola 71), the mixed strategy precedence of XPL (MHW 70), and Wirth-Webber simple precedence (WW 66). The Toronto comparisons are relevant to LIS because, for the cases considered, the resolution of the CFSM inadequate states by the LALR(k) algorithm is equivalent to the resolution by the CLR(k) algorithm. Leaving out the details, we reproduce their results in Figures IV.1 and IV.2. As indicated, the LALR(k) strategy is significantly more efficient, both in space and time, than the precedence methods. The important result of their investigations, however, is not the degree to which LALR(k) is more efficient, but that they compare "very favorably in efficiency with precedence methods which have themselves proved to be quite acceptable in practice. We conclude that efficiency is not an objection to LR(k)-based techniques".

IV.B.2 CLR(k) Primary Parser Efficiency

In this section, we report on the empirical measurements of the space and time efficiency of selected CLR(k) parsers produced by LIS. The measurements were taken on Multics (H645) when the system was simultaneously supporting 20 users, and configured with one central processing unit and 384,000 words (36 bits/word) of main memory. The LIS Processor Control, which included the

Grammar	Vocabulary Size Terminals	Non-Terminals	Number of Productions	MSP Bytes	WMSP Bytes	LALR Bytes	LALR States
XPL	42	49	109	3274	*	1250	234
EULER	78	44	120	3922	4321	1606	223
EULER-<number>	65	39	100	3017	3204	1276	192
ALGOL 60	62	82	173	>6800**	>6100*	2821	376

* Not a WMSP grammar
** Not an MSP grammar

<u>Toronto Space Comparisons</u>

Figure IV.1

Program	Size		Number of Reductions	MSP Seconds	LALR Seconds
	Cards	Tokens			
compactify	77	439	1,262	0.84	0.52
XCOM	4,241	24,390	66,108	45.35	25.11
DOSYS	7,291	29,334	81,581	55.58	30.49
DIAL	6,504	32,136	116,803	58.24	32.65

<u>Toronto Speed Comparisons</u>

Figure IV.2

control for lexical analysis, occupied 905 words. In measuring time efficiency, programs containing not less than 12,000 tokens were used, and measurements were taken over several executions and the results averaged.

The languages for which we report our results are the primary grammars of FILETRAN, SCHEMA, and PL/I (see discussions in Section IV.E). Our measurements of DPDA size exclude the requirements of the key-symbol table of the associated grammar.

a. FILETRAN

Grammar:	337 Productions
	135 Non-Terminals
	169 Terminals
DPDA Size:	855 States
	1553 Words
Parse Speed:	145,000 Tokens/Minute

b. SCHEMA

Grammar:	432 Productions
	184 Non-Terminals
	99 Terminals
DPDA Size:	807 States
	1520 Words
Parse Speed:	100,000 Tokens/Minute

c. PL/I

Grammar:	358 Productions
	139 Non-Terminals
	135 Terminals
DPDA Size:	768 States
	1717 Words
Parse Speed:	90,000 Tokens/Minute

As with the Toronto comparisons, the essential point of these results is that the parsers produced by LIS are quite acceptable on efficiency criteria.

IV.B.3 CLR(k) Lexical Parser Efficiency

We have been successful in adapting the CLR(k) parsers produced by LIS to the production of efficient lexical parsers. This has been possible because of several factors:

a. The phrase structure of lexical constructs is generally of little interest, the task of lexical analysis being limited to the efficient recognition of rather simple constructs.

b. The constructs defined by the lexical grammar typically include sets of structurally equivalent terminal characters, such as the set of the lower case letters, and the set of the integers from zero to nine.

c. The sets of structurally equivalent terminal characters have consecutive numeric character code representations on most computers.

Taking advantage of these factors, we were able to modify the DPDAs and the lexical parser control to admit read transitions and look-ahead transitions of the form, S->F. The parsing interpretation of such a transition is that a terminal character whose numeric value lies between the numeric values of S and F (inclusively) satisfies the condition of the transition. Furthermore, when a character has been found that satisfies the condition, all subsequent characters satisfying the condition are also accepted prior to taking the transition. The utilization of these "special lexical encodings" in developing lexical grammars and their parsers is discussed in Appendix A.

A rather large lexical grammar is the lexical grammar of PL/I given in Appendix D. The following empirical measurements of the efficiency of the PL/I lexical parser produced by LIS were taken on Multics under the same conditions that prevailed during the measurements discussed in the previous section.

Grammar:	36 Productions
	11 Non-Terminals
	73 Terminals
DPDA Size:	73 States
	136 Words
Parse Speed:	80,000 Characters/Minute

The space efficiency of the parser is quite good. While the time efficiency is certainly acceptable, there exist additional optimizations that may be performed to increase this efficiency even further. First, the key-symbol table is presently searched linearly, so that hashing of the table will result in significant increases in lexical parsing speed. Second, we can perform optimizations on the DPDA which will eliminate apply states that apply unit productions (Pag 73). Finally, on a computer such as the IBM System/360 or the IBM System/370, we can translate the entire lexical parser into an assembly language program, and employ the highly efficient translate and test instruction in implementing the read and look-ahead states. This optimization will be at a slight expense in space efficiency, but will enable the speed of CLR(k) lexical parsers to compare favorably with the best hand coded assembly language alternatives.

IV.C Syntax Directed Error Detection, Reporting, and Recovery

Users of a particular artificial language will inevitably make (context-free) syntax errors in the text submitted for processing. In response to such errors, the language processor must be able to detect the errors, report the errors in an intelligible form, and recover from the errors in such a way that processing may continue. These error handling procedures must be localized to the input text immediately surrounding a particular error, so that recovery from that error will not result in skipping over large portions of the input text. Furthermore, the error handling procedures should have appropriate termination conditions so as not to produce an avalanche effect when particularly difficult errors are encountered. In this section, we discuss the basic design of the error handling procedures planned for LIS. The design that we outline is an extension of the work by James (Jam 71), which is, in turn, an extension of the work done by Leinius (Lei 70) on syntax directed error handling.

IV.C.1 Error Detection

Error detection in the CLR(k) parser presented in Section III.F is straightforward: an error exists in the input text whenever the parser enters a read state and the current input symbol does not match any of the transition symbols of the state. Were it not for the default transition optimization that is performed on look-ahead states, errors would be detected in these states as well. However, because of the optimization, errors that would ordinarily be detected in a look-ahead state will go undetected until the next read state is entered. In the meantime, the parser may have entered apply states, resulting in the execution of semantics and the popping of the DPDA state stack. The non-deterministic execution of semantics causes problems for subsequent language processing, while the stack popping complicates the recovery procedures. Our solution, therefore, is not to perform the default look-ahead transition optimizations, so that the resulting CLR(k) parser will detect context-free syntax errors "as soon as they occur".

The ability of the CLR(k) parser to detect errors "as soon as they occur" is a significant one, since there are many parsing schemes in which this ability does not exist. Thus, for example, in the case of the precedence methods, it is entirely possible for consistent precedence relations to exist within a handle that does not match the right side of any production, and which, therefore, contains an error.

Error detection, reporting, and recovery become significantly more complex as a result (Len 70). The detection capability of the CLR(k) parser is therefore of significant value, since it properly initiates the error handling procedures.

IV.C.2 Error Reporting

The minimum information that should be delivered as an error diagnostic includes the point in the input text at which the error was detected, the symbol encountered at that point, and the set of symbols that could legitimately be accepted at that point. It should be obvious that this information is available, given the detection capability discussed in the previous section. However, in addition to this basic information (based strictly on the language's terminal symbols), it is also desirable to deliver information on the phrase structure surrounding the detected error. This information can be delivered if the non-terminal transitions are retained in the DPDA. Reporting of the phrase structure up to the point of the error would then be accomplished by simply going down the DPDA state stack and reporting the symbols (terminal and non-terminal) that access the stacked states. Likewise, the non-terminal transitions may be used to suggest possibilities for acceptable phrases beginning at the point of the error. We conclude, therefore, that proper error reporting can be automatically performed for the CLR(k) parser, but only if the non-terminal transitions are retained in the DPDA. So as to maintain the space efficiency of the parser, however, we elect to enter this non-terminal transition information into a separate "error handling" segment, to be referenced only when handling errors.

IV.C.3 Error Recovery

Having developed procedures to handle automatically the task of context-free syntax error detecting and reporting, we now turn our attention to the problem of error recovery. We incorporate two algorithms for error recovery into our CLR(k) error handling procedures. In the first, detection of an error causes a check to be made to determine whether the error is likely the result of a key-symbol spelling error. This could occur in the parse at a point at which a certain set of key-symbols is acceptable, but at which an identifier (in the usual sense) or an unacceptable key-symbol is actually encountered. In such a situation, we use well developed spelling comparison algorithms (Mor 70) to compare the encountered symbol against the possible key-symbol transitions of the current state. If the comparison is positive for exactly one of the transitions, a spelling correction diagnostic is issued, the transition is taken, and the parse continues. Otherwise, the recovery attempt proceeds to the second recovery algorithm.

The second error recovery algorithm is based on the phrase structure of the input text, and is thus considered a phrase structure recovery algorithm. The algorithm first isolates the portion of the input text containing the error. It relies on the error reporting procedure discussed in the previous section to identify the phrase that was in the process of being parsed when the error was detected. The basic approach of the algorithm is to search down the DPDA state stack looking for a state containing a transition on a non-terminal to which the partial phrase could be reduced. Given such a state, it takes the non-terminal transition satisfying the condition and continues the parse until it leads to a DPDA read state. It then scans the input text until a terminal symbol is found that matches one of the transition symbols from the read state. The intervening text is skipped over, and the parse is resumed.

The above recovery algorithm is in need of refinement before it can be considered operational. The first refinement has to do with the way in which the algorithm determines the possible non-terminals to which the partial phrase may be reduced. Only certain non-terminal transition symbols of certain states may be considered candidates. For a particular stacked state, the candidate non-terminal transition symbols are determined as follows:

> The set of candidate non-terminal transition symbols of the state is initialized to those that are defined by productions whose first symbol is the terminal symbol read by the parser when in that state. If no such non-terminal transition symbols exist, then the stacked state is not a candidate recovery state. Otherwise, the closure of the set is obtained by including all of the state's non-terminal transition symbols that are defined by productions whose first symbol is a member of the set.

There is a definite hierarchy to the set of non-terminal transition symbols associated with a particular terminal transition symbol of a particular DPDA read state. This hierarchy is computed for each terminal transition of each DPDA read state, and stored in the "error handling" segment mentioned in the last section. The second refinement to the basic recovery algorithm involves the utilization of this hierarchy. The hierarchy specifies a definite ordering to the the application the the recovery algorithm, an ordering that is necessary because, in general, there does not exist a unique recovery for each error. Utilizing this hierarchy, the procedure is to attempt recovery to the lowest possible non-terminal in the hierarchy that leads to a state that can read a terminal symbol within the (heuristically set) bounds of the erroneous phrase. However, due to the possibility of subsequent errors within the phrase, it may be appropriate to recover according to a higher level in the hierarchy or even to a lower state in the DPDA state stack.

In the case of a grammar in which the statements are delimited by a reserved symbol (such as ";" in PL/I), recovery will generally be accomplished within the statement containing the error. In all cases, the algorithm will terminate because the largest partial phrase can always be reduced to the goal symbol of the grammar (<primary_non_terminal>).

IV.D <u>LR(k) Hierarchy</u>

In this section, we define the significant levels in the LR(k) hierarchy and indicate the position within this hierarchy of the CLR(K) grammars.

The hierarchy of LR(k) grammars is indicated below. The grammars are listed (top to bottom) in order of decreasing grammarical comprehension.

LR(k)
LALR(k)
CLR(k)
SLR(k)
LR(0)

IV.D.1 <u>LR(k)</u> (Left to Right, k symbols)

A context-free grammar, G, is LR(k) if and only if every canonical form A = PB of G, except A = S (S is the goal symbol) has a unique characteristic string P#p which can be determined by investigating only P and the first k symbols of B.

IV.D.2 <u>LALR(k)</u> (Look Ahead Left to Right, k symbols)

A context-free grammar, G, is LALR(k) if and only if the inadequate states of G's CFSM can be resolved with k symbols of look-ahead. Operationally, this definition is of little use. To get an operational description, we turn to

Lalonde's discussion of the LALR(k) algorithm (Lal 71):

The philosophy of the LALR routine is such that when a state is inadequate, a tree of predecessor states which can access this inadequate state is built up one level at a time (one level meaning one transition). To each level there corresponds a predecessor set. The depth or numer of successive predecessor sets which must be calculated depends naturally on the maximum number of states which must be pulled (by applying a production) starting originally from the inadequate state. These predecessor sets can therefore be considered as forming a mainline predecessor path (in actual fact, a predecessor tree) which dictates the past history up to this particular inadequate state. From any given state, if no #-symbols are encountered, it is an easy matter to project forward to obtain sequential lists of k terminals which could be seen by look-ahead. When a #(P) symbol is encountered, however, the terminals which could follow if production P were applied must be collected. To do this, the number of states equal to the length of the RHS of production P is pulled. Furthermore, of the possible states which are now visible, only those which can reach the inadequate state (namely those on the mainline predecessor path) are candidates. Having found the production goal for production P in each of these states, the process of collecting terminals is resumed starting from each of the destination states of these production goals. These terminals are of course added to the successive terminals collected so far. This is obviously very recursive and repetitive but nevertheless, essential for localized look-ahead.

Moreover, if during look-aheads, side branches are followed which lead away from the mainline predecessor path (this occurs whenever a symbol is added to the set and at least one other succeeding symbol is sought), then pulls which are performed there must fall on the side branch taken or (if the pull has enough depth) on the mainline predecessor path.

In all cases, the mainline predecessor path is backed up dynamically as far as necessary.

The above discussion does not treat the termination condition adequately because it does not specify the action to be taken if a particular CFSM state is re-entered when looking for a symbol at a given look-ahead level. By private communication, Lalonde indicated to this author that his algorithm does not perform such re-entry, the assumption being that re-entry would add no new look-ahead symbols. This author implemented Lalonde's algorithm and verified a suspicion that this re-entry assumption is invalid. Empirically, this conclusion is based on the failure of the algorithm to properly handle the PL/I <conditional_statement> as given in Appendix D. A sufficient condition for rejecting re-entry for a particular state and look-ahead level is that the mainline predecessor paths and the side paths for the proposed re-entry be the same as those existing at the time of the original entry. However, the magnitude of the data manipulations associated with saving and comparing predecessor paths and side paths influenced us to reject the LALR(k) algorithm in favor of the simpler, yet comprehensive, CLR(k) algorithm.

IV.D.3 <u>CLR(k)</u> (<u>C</u>omprehensive <u>L</u>eft to <u>R</u>ight, <u>k</u> symbols)

A context-free grammar is CLR(k) if and only if the inadequate states of its CFSM can be resolved by the algorithm presented in Section III.D.3. The basic difference between the CLR(k) algorithm and the LALR(k) algorithm is that the CLR(k) algorithm takes all apply transitions of the apply states entered during look-ahead. This alleviates the need to save mainline predecessor paths and side paths and guarantees that re-entry (in the previous sense) need never be performed. The CLR(k) grammars represent a subset of the LALR(k) grammars, although the only grammars that have been found to be LALR(k) and not CLR(k) have been pathological ones.

A personal note is appropriate regarding the definition of the CLR(k) grammars. The CLR(k) grammars were defined by Frank DeRemer and this author out of a basic dissatisfaction with the overhead involved in the LALR(k) algorithm and because we believed that the CLR(k) algorithm would cover virtually all grammars of "practical" interest. Thus far, this has proved to be a reasonable tradeoff.

IV.D.4 <u>SLR(k)</u> (<u>S</u>imple <u>L</u>eft to <u>R</u>ight, <u>k</u> symbols)

A context-free grammar is SLR(k) if and only if the inadequate states of its CFSM can be resolved using the SLR(k) algorithm (DeR 69). The SLR(k) algorithm is similar to the CLR(k) algorithm except that when a CFSM apply transition is encountered during look-ahead, a computation is made on the <u>grammar</u> to determine the set of terminal symbols that can legitimately follow the non-terminal defined in the production being applied. The SLR(k) condition is thus based on information <u>global</u> to the grammar, and is therefore not as comprehensive as the LALR(k) and the CLR(k) conditions, which are based on information <u>local</u> to the CFSM inadequate state being resolved.

IV.D.5 <u>LR(0)</u> (<u>L</u>eft to <u>R</u>ight, <u>0</u> symbols)

A context-free grammar is LR(0) if and only if its CFSM contains no inadequate states. Virtually no grammars of "practical" value are LR(0).

IV.E Applications of LIS

In this section, we consider some of the language/processor developments in which the Language Implementation System has been utilized. Our discussions here will be brief; the reader wishing more detail on applications of LIS is referred to Appendices A, C, D, and E. The applications discussed here are only representative of those to date, which also include PAL, Algol 60, and the Procedure Division of COBOL.

IV.E.1 assign

The assign language is a simple assignment statement language. The processor that we developed for assign is a simple translator from the language into a symbolic intermediate language, the type that may be produced by the interpretation phase of a compiler, for example. The assign language is discussed in detail in Appendix A, Section A.7. We mention the assign language here because it is probably the simplest example of the way in which the syntax and semantics of an artificial language may be integrated into a concise language definition for processing by LIS. As such, it serves as a good introduction to the reader wanting to investigate the more comprehensive applications in the other appendices.

IV.E.2 p16535

The p16535 language is a block structured language that illustrates the translation of fundamental high level language constructs into a particular formal semantic system. The p16535 language and its processor are discussed in detail in Appendix C. p16535 was developed as a term project for a graduate computer science course at MIT, and the author was ably assisted in this development by fellow graduate students, Thomas Gearing and Gordon Weekly. (AGW 72).

IV.E.3 FILETRAN

The FILETRAN language is being developed at Honeywell for the purpose of providing a facility for translating arbitrary data files into data files compatible with a particular Honeywell computer. The size and efficiency of the FILETRAN language and its processor are indicated in Section IV.B.2.

IV.E.4 SCHEMA

The COBOL SCHEMA and its processor represent an implementation of a data base schema language. The size and efficiency of the SCHEMA language and its processor are indicated in Section IV.B.2.

IV.E.5 PL/I

The primary grammar of PL/I that appears in Appendix D is the most complex grammar submitted to LIS to date. The grammar is a very large sub-grammar of the IBM Laboratory Vienna's specification of the concrete syntax of PL/I (AOU 68), and includes declarations, input/output, and on-conditions.

Appendix D also includes the PL/I lexical grammar, which is the most complete lexical grammar yet submitted to LIS.

IV.E.6 express

The express language is the only example that we give in which LIS was utilized in the development of an interactive management information/decision system language. A discussion of this development is given in Appendix E.

IV F Areas for Future Research and Development

In this section, we briefly consider areas of research and development that we feel will have significant impact on language implementation system technology. Our discussions here are admittedly too abbreviated to do more than suggest the broad boundaries of these efforts.

IV.F.1 Formal Semantic Systems

It is widely recognized that formal semantic systems have not achieved the same level of development as the corresponding work in formal grammar systems and the application of automata theory to the recognition of artificial languages. As Winograd states (Win 71), "The field of semantics has always been a hazy swampland". This is due to several factors. First, the problems of formal semantics are simply more difficult than the problems of language recognition. Second, the rapid development of fundementally new language constructs and hardware features has made the specification of a set of formal semantic primitives even more difficult. Finally, and basicly because of the previous two points, there seems to be a general disagreement as to what constitutes an appropriate set of formal semantic primitives. Nevertheless, progress continues to be made, as witnessed by the substantial amount of published material on the subject (see Bibliography). Appendix D describes our initial attempt at utilizing LIS in a way that will contribute to the advancement of formal semantic systems. We see two areas of future development that are appropriate to the approach that we have taken. First, the implementation of a Base Language interpreter would provide an experimental environment in which the Base Language primitives could be challenged and modified, the criteria being their ability to adequately and conveniently represent higher level language constructs.

Second, our work on p16535 and its Base Language translator has convinced us that PL/I is a most unattractive language for specifying formal semantics. Therefore, an appropriate next step would be the utilization of LIS in developing a language well suited to specifying the Base Language interpretation of higher level language constructs.

IV.F.2 Grammar Complexity Measures

It has been widely accepted on intuitive grounds that there exist significant variations in the complexities of popular programming languages. Few would argue, for example, that the structure of the Basic language is rather simple, while that of PL/I is relatively more formidable. We conveniently express this notion by saying that PL/I is a more complex language than Basic. But what does complexity mean? Can it be measured? Recent investigations into complexity measures (Don 72, Hag 70) are beginning to uncover some of the issues, but as yet, no satisfactory measures have been proposed. With respect to a particular language, possible suggestions for complexity measures include the difficulty in recognizing the language's constructs (time of parse), the number of steps in the derivations of the constructs, the size of the parse tree, and the space requirements of the parser for the language. On first examination, the above issues would seem to have little relevance to the average programmer. However, linguistic constructs that are complex in a formal sense (length of derivation, time of parse, etc.) are also complex in a human sense - they take a long time to write, to understand, and to debug. Thus, complexity is significant both from a grammar-theoretic viewpoint and from a practical viewpoint. Furthermore, to the extent that measures of complexity can be appropriately defined, they may be utilized in restructuring languages to reduce complexity, thereby making them more palatable.

Although our experience with LIS is yet too limited to enable us to define precisely a formal set of complexity measures, we have nevertheless witnessed significant variations in "complexity" among the languages to which LIS has been applied. We briefly consider two of these, and in so doing, suggest possibilities for future developments in the evolution of complexity measures, based on LR(k) systems.

In Section IV.B.2, we discussed the efficiency of two of the languages to which LIS has been applied, FILETRAN and PL/I. Referring again to that discussion, we note that while the sizes of the grammars and their associated DPDAs are roughly equivalent, a vast difference exists in the rate at which the two languages can be recognized by our CLR(k) parser. The fact that FILETRAN can be parsed at 145,000 tokens/minute, while PL/I is parsed at the relatively slower rate of 90,000 tokens/minute indicates that PL/I presents a more difficult recognition problem. We shall accept one of the above suggestions and associate difficulty of recognition with grammar complexity. We are, therefore, interested in a predictor of complexity based on our CLR(k) strategy. One predictor that we have observed is the time taken to compute the CLR(k) parser of a grammar. Relatively speaking, the longer the time taken to compute the parser, the longer will be the time required to parse the constructs of the language. In the present case, we note that the parser for PL/I took almost twice as long to compute as did the parser for FILETRAN. Another predictor that we have observed is the number of look-ahead states in the resulting parser. Again, relatively speaking, the more look-ahead states, the longer the required time to recognize. This is reasonable, since the need for look-ahead originates from a local ambiguity, a basic form of complexity. In our present case, the parser for PL/I contained 60 look-ahead states, while the parser for FILETRAN contained only 23 look-ahead states. The look-ahead states of the PL/I parser are largely due to the ubiquitous <expression>.

IV.F.3 Microprogrammed LIS Processor Control

The space and time efficiency of our CLR(k) parsers have been amply demonstrated in Section IV.B. Furthermore, owing to the simplicity of the CLR(k) parser control, it is entirely feasible to implement this control directly in hardware, resulting in even greater parse time efficiency. Given the trends in hardware technology, it is appropriate to consider implementing this control by microprogramming. Whether this effort is undertaken on a particular computer is dependent on such factors as the increase in efficiency versus cost, and the number of language processors that would benefit from the increased efficiency.

In a paper for a graduate computer science course at MIT (Alt 72a), the author investigated some of the issues involved in a firmware implementation of CLR(k) parser control. The actual representation of the control and the control primitives was in the form of a computation schema (Den 70). Transformations can be performed on the schema to yield a data flow structure and a control structure, which together constitute an asynchronous modular hardware representation of CLR(k) processor control.

IV.G Conclusion

Our objective in this thesis was to develop a language implementation system satisfying the criteria established in Chapter I, and to utilize this system in the development of a wide variety of artificial languages and their associated processors. We feel that we have been successful in this regard. We have shown the Language Implementation System to be an efficient, reliable, and flexible language development and implementation support system. We have demonstrated the system's applicability, not only to traditional languages and their processors, but also to problem oriented languages and interactive languages for management information/decision systems.

It is the author's conviction that the demand for special purpose - end user computational systems and interactive decision support systems will accelerate rapidly over the coming years. Furthermore, it is also believed that the successful development and evolution of such systems will depend critically on the appropriateness of the supporting language facilities. We are, therefore, convinced that systems such as the Language Implementation System will come to play a major role in the expansion of the domain to which computation can be successfully applied.

Bibliography

AGW 72	Altmanv, V.E., Gearing, T., and Weekly, G.: The Design and Implementation of a Translator from a Block Structured Language into the Common Base Language. unpublished term paper, MIT: Cambridge, Mass. (1972)
AOU 68	Alber, K., Oliva, P., and Urschler, G.: Concrete Syntax of PL/I. Tech. Rept. TR 25.084, IBM Laboratory Vienna: Vienna, Austria (1968)
AU 72	Aho, A.V. and Ullman, J.D.: The Theory of Parsing, Translation, and Compiling: Vol. 1 Parsing, Vol. 2 Compiling. Prentice-Hall, Inc.: Englewood Cliffs, N.J. (1972)
Alt 71	Altman, V.E.: A Language Implementation System. S.M. Thesis Proposal, MIT: Cambridge, Mass. (1971)
Alt 72a	Altman, V.E.: A Computation Schema Representation of CLR(1) Parser Control. unpublished term paper, MIT: Cambridge, Mass. (1972).
Alt 72b	Altman, V.E.: LIS User Reference Manual. Honeywell Information Systems, Inc.: Waltham, Mass. (1972)
Ch 56	Chomsky, N.: Three Models for the Description of Language. IREE Trans. Inf. Th., Vol. IT2 (1956) 113 - 124
Ch 59	Chomsky, N.: On Certain Formal Properties of Grammars. Information and Control 2 (1959) 137 - 167
Ch 63	Chomsky, N.: Aspects of the Theory of Syntax. MIT Press: Cambridge, Mass. (1963)
DeR 69	DeRemer, F.L.: Practical Translators for LR(k) Languages. Project MAC Tech. Rept. TR 65, MIT: Cambridge, Mass. (1969)
DeR 70a	DeRemer, F.L.: A Proposed Translator Writing System Language. CEP Report Vol. 3, No. 1, Univ. of California: Santa Cruz, Ca. (1970)

Preceding page blank

DeR 70b DeRemer, F.L.: Simple LR(k) Grammars: Definition and Implementation. CEP Report Vol. 2, No. 4, Univ. of California: Santa Cruz, Ca. (1970)

DeR 71 DeRemer, F.L.: Simple LR(k) Grammars. CACM 14,7 (July 1971) 453 - 460

Den 69 Dennis, J.B.: Programming Generality, Parallelism, and Computer Architecture. Information Processing 68. North-Holland: Amsterdam (1969) 484 - 492

Den 72 Dennis, J.B.: On the Design and Specification of a Common Base Language. Project MAC Tech. Rept. TR-101, MIT: Cambridge, Mass. (1972)

Don 72 Donovan, J.J.: Systems Programming. McGraw-Hill, Inc.: New York (1972)

Ear 70 Earley, J.: An Efficient Context-Free Parsing Algorithm. CACM 13,2 (Feb. 1970) 94 - 102

FG 68 Feldman, J.A. and Gries, D.: Translator Writing Systems. CACM 11,2 (Feb. 1968) 77 - 113

Fli 72 Flinker, E.: Translation of a Block Structured Language into the Common Base Language. Project MAC Computation Structures Group Memo. 66, MIT: Cambridge, Mass. (1972)

GP 65 Griffiths, T.V. and Pettrick, S.R.: On the Relative Efficiencies of Context-Free Grammar Recognizers. CACM 8,5 (May 1965) 289 - 299

Gin 62 Ginsburg, S.: An Introduction to Mathematical Machine Theory. Addison-Wesley, Inc.: Reading, Mass. (1962)

Gin 66 Ginsburg, S.: The Mathematical Theory of Context-Free Languages. McGraw-Hill, Inc.: New York (1966)

Gor 71 Gorrie, J.D: A Processor Generator System. Tech. Rept. CSRG-3, Univ of Toronto: Toronto, Canada (1971)

Gr 71 Gries, D.: Compiler Construction for Digital Computers. John Wiley and Sons, Inc.: New York (1971)

HU 69	Hopcroft, J.E. and Ullman, J.D.: Formal Languages and their Relation to Automata. Addison-Wesley, Inc.: Reading, Mass. (1969)
Hag 70	Haggerty, J.P.: Complexity Measures for Language Recognition by Canonic Systems. Project MAC Tech. Rept. TR 77, MIT: Cambridge, Mass. (1970)
Hen 68	Hennie, F.C.: Finite State Models for Logical Machines. John Wiley and Sons, Inc.: New York (1968)
HoLa 71	Horning, J.J. and Lalonde, W.R.: Empirical Comparison of LR(k) and Precedence Parsers. Tech. Rept. CSRG-1, Univ. of Toronto: Toronto, Canada (1970)
IPS 72	Interactive Planning Systems User Guide. Interactive Planning Systems, Inc.: New York (1972)
Jam 71	James, L.R.: A Syntax Directed Error Recovery Method. M.S. Thesis, Univ. of Toronto,: Toronto, Canada (1971)
Kn 65a	Knuth, D.E.: On the Translation of Languages from Left to Right. Inf. Control 8 (Oct. 1965) 607 - 639
Kn 65b	Knuth, D.E.: Semantics of Context-Free Languages. Math. Sys. Th. 2 (Feb. 1965) 127 - 145
Kor 69	Korenjak, A.: A Practical Method for Constructing LR(k) Processors. CACM 12,11 (Nov. 1969) 613 - 623
LLS 68	Lucas, P., Lauer, P., and Stigleitner, H.: Method and Notation for the Formal Definition of Programming Languages. Tech. Rept. TR 25.087, IBM Laboratory Vienna: Vienna, Austria (1968)
LW 69	Lucas, P. and Walk, K.: On the Formal Description of PL/I. Annual Review in Automatic Programming: Vol. 6, Part 3. Pergamon Press: London (1969) 105 - 182
La 71	Lalonde, W.R.: An Efficient LALR Parser Generator. Tech. Rept. CSRG-2, Univ. of Toronto: Toronto, Canada (1971)
Lau 68	Laur, P.: Formal Definition of Algol 60. Tech Rept. TR 25.088, IBM Laboratory Vienna: Vienna, Austria (1968)

Lei 70 Leinius, R.P.: Error Detection and Recovery for Syntax Directed Compiler Systems. Ph.D. Thesis, Univ. of Wisconsin: Madison, Wisconsin (1970)

Lu 68 Lucas, P.: Two Constructive Realizations of the Block Concept and Their Equivalence. Tech. Rept. TR 25.085, IBM Laboratory Vienna: Vienna, Austria (1968)

MHW 70 McKeeman, W.M., Horning, J.J., and Wortman, D.B.: A Compiler Generator. Prentice-Hall, Inc.: Englewood Cliffs, N.J. (1970)

Mc 66 McCarthy, J.: A Formal Description of a Subset of Algol. Formal Language Description Languages for Computer Programming. North-Holland: Amsterdam (1966) 1 - 12

Min 68 Minsky, M.(ed): Semantic Information Processing. MIT Press: Cambridge, Mass. (1968)

Mor 70 Morgan, H.L.: Spelling Correction in Systems Programs. CACM 13,2 (Feb. 1970) 90 - 94

Pag 73 Pager, D.: On Eliminating Unit Productions from LR(k) Parsers. University of Hawaii (1973)

RR 64 Randell, B. and Russell, D.J.: ALGOL60 Implementation. Academic Press: London (1964)

St 61 Steel, T.B.: UNCOL: The Myth and Fact. Annual Review in Automatic Programming: Vol. 2. Pergamon Press: London (1961) 325 - 344

Win 71 Winograd, T.: Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. Project MAC Tech. Rept. TR 84, MIT: Cambridge, Mass. (1971)

WW 66 Wirth, N. and Webber, H.: EULER: A Generalization of ALGOL and its Formal Definition. CACM 9,1 and CACM 9,2 (Jan. and Feb. 1966)

Appendix A

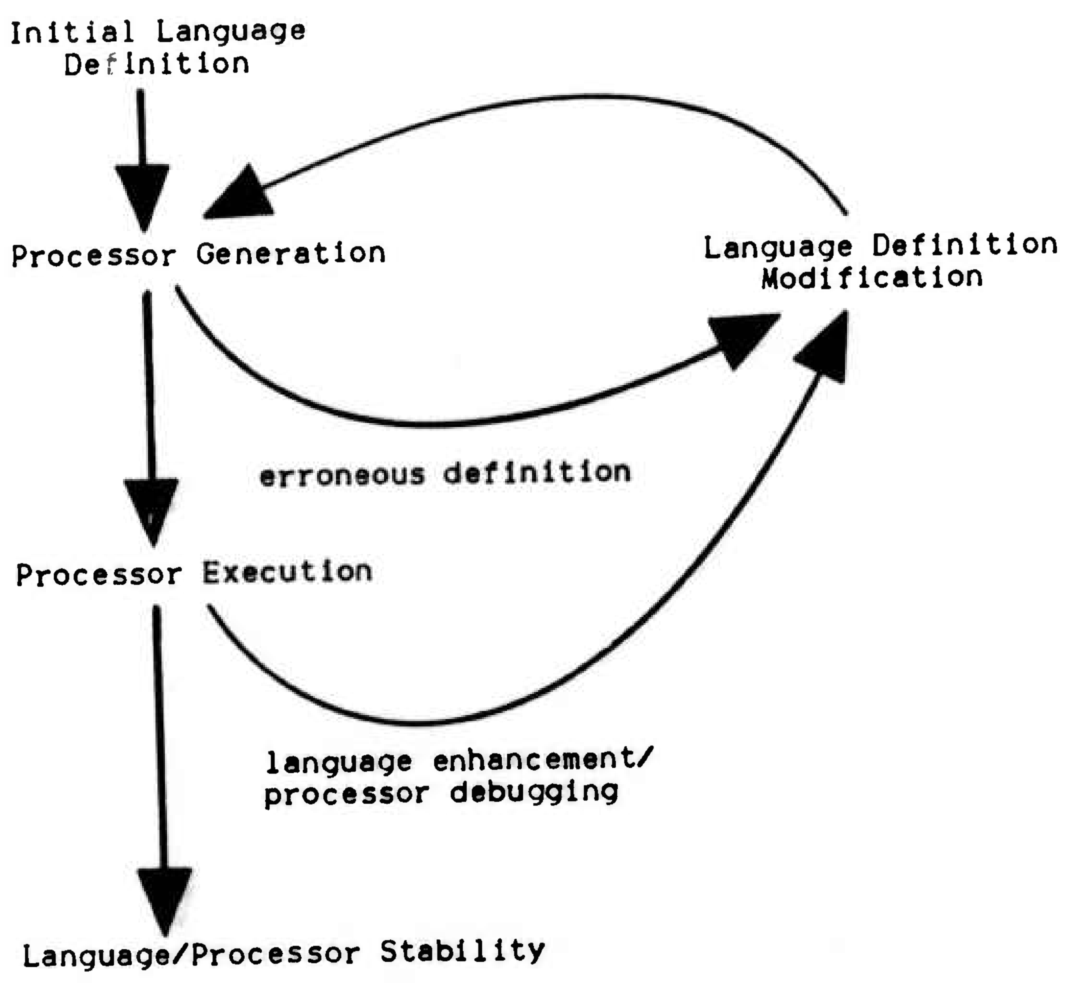
LIS User Reference Manual

A.1 Introduction

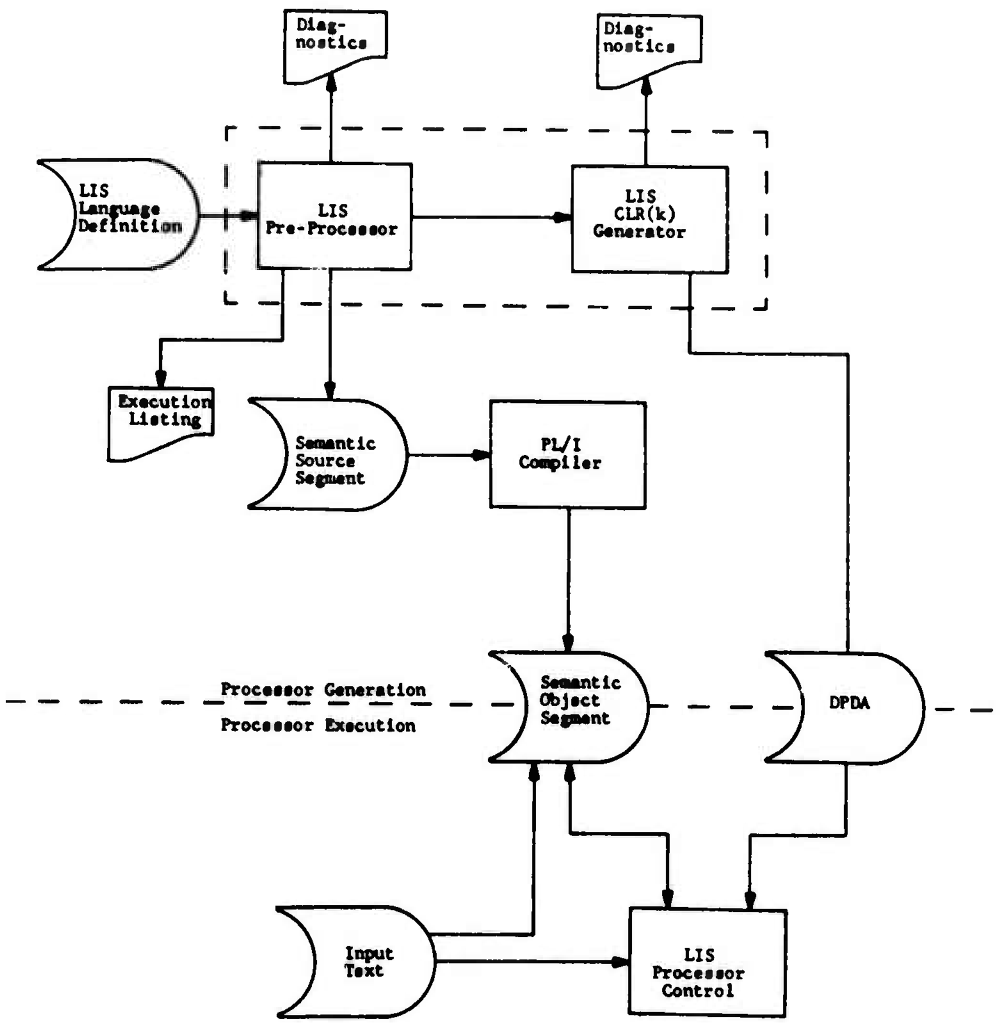
In this appendix, we present the LIS User Reference Manual. This Manual is an abridged form of a Honeywell publication of the same title (Alt 72b). Its purpose is to describe the way in which the language designer/implementer utilizes the Language Implementation System in developing artificial languages and their associated processors. The Manual is intended to be self-contained, and therefore includes certain discussions from Chapters I - IV that are relevant to the utilization of LIS.

A.2 <u>Language/Processor Development Using LIS</u>

The fundamental structure of The Language Implementation System is indicated in Figure A.1. Language Development resolves into two interacting phases, Processor Generation and Processor Execution:


Initial Language Definition
Processor Generation
Language Definition Modification
erroneous definition
Processor Execution
language enhancement/ processor debugging
Language/Processor Stability

<u>Language/Processor Development Using LIS</u>



Structure of The Language Implementation System

Figure A.1

A.2.1 Processor Generation

Processor Generation consists of the execution of the LIS Pre-Processor and the LIS CLR(k) Generator for purposes of computing the following functional results from the submitted LIS Language Definition:

a. The parsing tables (DPDAs) which are used to "drive" LIS Processor Control in parsing legal Input Text of the language.

b. A PL/I procedure which represents the semantic interpretation to be associated with the language's syntactic constructs.

LIS Language Definition

A precise specification of the format of an LIS Language Definition may be found in Sections A.3 and A.4. For our present purposes, however, we may consider an LIS Language Definition to consist of:

a. A Backus Naur Form specification of the syntax of the language being defined.

b. A PL/I specification of the semantics of the language, expressed in-line with the BNF specification on a per-BNF rule basis. In specifying the semantics of a particular syntactic construct, the language designer/implementer uses PL/I to define the actions that his language processor is to perform when the corresponding syntactic construct is recognized.

LIS Pre-Processor

The LIS Pre-Processor performs the following functions:

a. The LIS Pre-Processor computes the Semantic Source Segment from the Language Definition.

b. The LIS Pre-Processor performs various validity checks and analysis procedures on the submitted grammar, delivering diagnostics for those checks and analysis procedures that the grammar fails to satisfy. Certain of the checks and procedures are of a warning nature only: failing to satisfy these will not prevent the activation of the LIS CLR(k) Generator. Others are of a fatal nature, and <u>must</u> be satisfied if the LIS CLR(k) Generator is to be activated. The checks and analysis procedures that have been implemented on LIS are discussed from the language design viewpoint in Section A.3.

<u>LIS CLR(k) Generator</u>

If the LIS Pre-Processor encounters no fatal errors in the LIS Language Definition, control is automatically passed to the LIS CLR(k) Generator. This phase of the system attempts to compute a CLR(k) parser for k less than or equal to a certain internally set value (currently set at 3). The CLR(k) (<u>C</u>omprehensive <u>L</u>eft to <u>R</u>ight, looking ahead a maximum of <u>k</u> symbols) grammars constitute a large subset of the LR(k) grammars, which in turn possess the following characteristics:

a. The LR(k) condition generates exactly the deterministic context-free grammars.

b. The LR(k) grammars represent the largest class of grammars known to be parsable in linear time (proportional to the length of the input text) during a single left to right scan.

c. A grammar satisfying the LR(k) condition is unambiguous.

Intuitively, the LR(k) condition implies that the identity of a particular syntactic construct may be ascertained by looking indefinitely far to the left and at most k symbols to the right of the current position in the parse (symbols meaning characters or lexical constructs, depending on whether a lexical or primary grammar is being defined, respectively). This is an extremely comprehensive condition, and covers virtually all artificial languages that are likely to be of "practical" interest.

In attempting to compute the parsers, the LIS CLR(k) Generator delivers diagnostics for those areas of the language that do not satisfy the CLR(k) condition. These diagnostics include sufficient information on the language's local ambiguities to enable the language designer/implementer to modify the syntax of his language in order to make it CLR(k). Assuming that the grammar is CLR(k), the functional output of the LIS CLR(k) Generator is a segment containing one or two DPDAs, depending on whether a lexical parser, a primary parser, or both, are computed. The DPDAs, in combination with LIS Processor Control, constitute the parsers for the processor of the language being defined.

A.2.2 Processor Execution

A processor for the artificial language specified by the LIS Language Definition is synthesized by combining the DPDAs and the Semantic Object Segment with LIS Processor Control.

LIS Processor Control coordinates the overall language processing activity. In parsing the Input Text, it is "driven" by the DPDAs, and upon recognition of a particular syntactic construct, it activates the semantics associated with that construct. It is the responsibility of the activated semantics subsequently to return control to LIS Processor Control so that language processing may continue.

The semantics can access the Input Text directly, and the normal situation is for Processor Control to coordinate these accesses by directing the semantics to specific text such as identifiers, key-symbols, etc. As indicated in Figure A.1, there is no explicit output from Processor Execution. It is therefore the responsibility of the semantics to manage its own output, as well as its alternate input files, temporary files, symbol tables, etc.

As a matter of processing efficiency, we note that it is possible to combine the Semantic Source Segment with LIS

Processor Control into a single segment, which is then compiled by the PL/I compiler. The advantage of this combination is that activations of BNF rule semantics may be effected by "goto"s rather than "call"s. In addition, an option is planned for LIS that will permit the DPDAs to be produced in the form of initialized PL/I declarations, which will also be compiled with LIS Processor Control. The combined result of these optimizations is that it will be possible for the three functional units comprising Processor Execution to be combined into a single PL/I procedure, resulting in significant improvements in space and time efficiency.

A.3 The Definition of Artificial Languages - Specification of Syntax

In this section, we discuss the way in which the language designer/implementer formulates the specification of the syntax of his language for processing by LIS. In Section A.4, we address the problem of semantic specificatiion. In following these discussions, the reader may find it useful to refer to the example of a simple language definition in Section A.7.

LIS provides the capability to compute both lexical and primary parsers from the appropriate specifications. For the most part, the structure and format of these specifications are the same. The features of the specifications that are grammar dependent are discussed in the appropriate sections below; the initial discussion will focus on the features that the specifications have in common.

A.3.1 Syntax Specification - General

The Language Implementation System accepts the syntax of artificial languages specified in free format Backus Naur Form (BNF). The purpose of the present discussion is to describe the structure and format of LIS acceptable BNF.

BNF Specification on LIS

On LIS, BNF specifications are structured as follows:

a. A BNF specification consists of a collection of BNF rules. With one minor exception (discussed in Section A.3.3), the order of the rules is irrelevant: the specification is non-procedural.

b. A BNF rule must start on a new line, may extend over several lines, and is terminated with an exclamation point ("!").

c. A BNF rule consists of a <left part> and a <right part>, separated by the string "::=".

Thus, a BNF specification is an unordered set of BNF rules of the following form:

<left part> ::= <right part> !

BNF Rule - <left part>

The <left part> of a BNF rule identifies the non-terminal that is defined in that rule. Thus, the format of the <left part> is identical to the format of BNF non-terminals, which is as follows:

"<"character-string">"

Character-string is restricted so that:

a. Character-string must not exceed 70 characters in length, including blank, tab, new-line, and new-page characters.

b. Character-string may not include any of the characters: "<", ">", ";", "!", or the string "::=".

BNF Rule - <right part>

The <right part> of a BNF rule consists of one or more *alternative* definitions of the <left part>, separated by the character, "¦". When referring to a particular alternative definition of the non-terminal <left part> in a multi-alternative BNF rule, one identifies the particular alternative in question or equivalently, to the *production* formed by constructing a single alternative BNF rule from the <left part> and that alternative. Alternatives exist primarily for convenient syntactic notation. However, when associating semantic interpretation with a particular BNF rule, account must be taken of the alternatives, and this is described in Section A.4.

Each alternative of a <right part> consists of at least one symbol. A symbol may be either a non-terminal or a terminal string of ASCII characters, the terminal string being subject to the following conditions:

a. The terminal string starts with the first ASCII character following the last symbol of the alternative (or following "¦" or "::=" if it is the first symbol of the alternative) that is neither a blank, nor a tab, nor a new-line, nor a new-page character. Since blank, tab, new-line and new-page characters are not normally considered part of a terminal string, they must be escaped if they are to be significant (escaping conventions are described in Section A.3.4).

b. The terminal string may not contain any of the characters "<", ">", "¦", "!", or the string

"::=", unless they are escaped.

A.3.2 Syntax Specification - Primary Grammar

We use the term, primary grammar, to refer to that subset of a particular language's complete syntax specification that excludes the specification of the lexical non-terminals of the language.

<primary non terminal>

The non-terminal, <primary_non_terminal>, has been reserved on LIS for purposes of identifying the goal symbol of the primary grammar. By defining <primary_non_terminal>, the user identifies the ultimate syntactic objective of his language. Except that it must be defined in order for LIS to compute a parser for the primary grammar, no fundamental limitations are placed on the definition of <primary_non_terminal>.

Spacing Conventions

The convention on LIS is that artificial languages are free format. The non-terminal, <non_lexical> has been reserved so as to permit the specification of those characters that are to serve as explicit delimiters and spacing characters within the defined language. The precise specification of <non_lexical> is given in Section A.3.4.

The interpretation of <non_lexical> is that an indefinite number of <non_lexical> characters may appear between those syntactic constructs of the defined language that correspond to the symbols in the alternatives of the BNF specification of the language. Equivalently, the interpretation to be associated with the delimiting of symbols in the alternatives of the primary grammar, is that an indefinite number of <non_lexical> characters may appear in the language at points corresponding to the primary grammar symbol delimiters. By an indefinite number of <non_lexical> characters, we mean zero or more, or one or more, depending on whether at least one <non_lexical> character is required in order to avoid conflict with a single string of characters that may be recognized as a particular lexical construct.

For example, consider the following BNF rule:

<go_to_phrase> ::= go to <identifier> !

Assuming that <identifier> is defined as usual (e.g. as in PL/I), we see that "gotoa" would be recognized as a single <identifier>, "goto a" would be recognized as a sequence of two <identifier>s, and that only some form of "go to a", in which at least one <non_lexical> character exists between each symbol, would be recognized as a <go_to_phrase>. On the other hand, consider the following example (<identifier> and <integer> are used in the usual non-terminal sense):

```
 <subscripted_identifier> ::= <identifier>  ( <integer> ) !
```

In this case, there is no possible conflict between <identifier> and "(", between "(" and <integer>, or between <integer> and ")", so that here an indefinite number implies zero or more. In the above rule, "(" and ")" are implicit delimiters: their use as delimiters is based on context and extracted from the grammar, as opposed to being stated explicitly, as with <non_lexical>.

In general, therefore, an indefinite number of <non_lexical> characters is interpreted to mean a number greater than or equal to that minimum number required to avoid improper recognition due to symbol conflicts with lexical constructs.

Although we have discussed the use of <non_lexical> as it applies to the primary grammar, it is the convention on LIS to define <non_lexical> with the specification of the lexical grammar in those cases in which the two grammars are defined in separate LIS Language Definitions.

A.3.3 Syntax Specification - Lexical Grammar

The lexical constructs of a language are those basic structural elements that are used in writing text in the language. These include the key-symbols, implicit delimiters (e.g. ")", "+", "-"), and the lexical non-terminals (e.g. <identifier> and <integer>, in the usual sense) that are built up from the terminal characters of the language, but whose substructure is of no fundamental interest, either syntactically or semantically. Of these three classes, the key-symbols and implicit delimiters are derived from the primary grammar, and it is the function of the lexical grammar to define the structure of the lexical non-terminals.

<lexical non terminal>

The non-terminal, <lexical_non_terminal>, has been reserved on LIS for purposes of identifying the goal symbol(s) of the lexical grammar, i.e. for identifying the lexical non-terminals of the language.

Definitions of <lexical_non_terminal> are restricted to the extent that its productions may consist only of single non-terminal symbols.

LIS may be executed for purposes of computing parsers for both primary and lexical grammars, or for either separately, and in each of these cases <lexical_non_terminal> has the following meaning:

a. Compute primary and lexical parsers
In this case, <lexical_non_terminal> is defined so as to establish the "division of labor" between the primary parser and the lexical parser. Thus, the lexical parser builds up the <lexical_non_terminal>s from the terminal characters, and the primary parser accepts the <lexical_non_terminal>s, key-symbols, and implicit delimiters as its basic elements in parsing the language being defined (as specified by <primary_non_terminal>).

b. Compute only the primary parser
This case is similar to case a, except that it is not necessary for LIS to have any immediate knowledge as to the structure of the <lexical_non_terminal>s. Thus <lexical_non_terminal> is defined so as to inform LIS of the basic elements that the primary parser will receive from the lexical parser.

c. Compute only the lexical parser
In this case, the definition of <lexical_non_terminal> serves to indicate to LIS that those non-terminals defined to be <lexical_non_terminal>s represent the goal symbols of the lexical grammar.

It may turn out to be convenient to call upon LIS twice, once to compute a parser for the primary grammar and again to compute a parser for the lexical grammar. Since the lexical grammar generally stabilizes long before the primary grammar, this activation sequence is efficient to the extent that changes may be introduced into the two grammars independently. In this situation, the definition of <lexical_non_terminal> performs a communication function between the two computations, and it is here that we have the only exception to the previously stated rule that the order of the BNF rules is of no significance:

> In those cases in which LIS is used to compute a primary parser and a lexical parser, but in *separate* activations using *separate* LIS Language Definitions, the system requires that the rules defining <lexical_non_terminal> be the same in each definition and that they appear (in the same order) as the first rules in each definition. However, if the separate activations are performed on the *same* definition segment, then the placement of the rules defining <lexical_non_terminal> is of no significance.

Special Lexical Encoding

A special lexical encoding convention has been implemented on LIS. Though implemented primarily for purposes of space and time efficiency in the resulting parsers, the convention also provides a convenient syntactic notation. The encoding permits the grouping of those characters that are equivalent in their effect on the structure of the <lexical_non_terminal>s, and that also have *continuous ASCII collating codes*. The encoding may be thought of as an additional type of symbol in the lexical grammar. The format of the encoding is as follows:

s->f

In the above format, *s* is the *s*tarting character in the sequence (the one with the smaller numeric code) and *f* is the *f*inal character in the sequence (the one with the larger numeric code). The encoding is represented as four contiguous characters, with no intervening blanks, tabs, etc. An example of the use of the encoding is as follows:

```
<small-letter> ::= a->z !
```

Non-Terminal Delimitation

The convention established for <lexical_non_terminal>s is that all characters not belonging to the defined constructs serve to delimit those defined constructs. This includes, of course, spacing characters such as blank, tab, etc. Furthermore, application of this convention is independent of any particular spacing within the alternatives defining the constructs, so that, for example, in the following definition of <identifier>, the fact that spacing exists between the symbols of the first alternative does not imply that spacing is permitted within <identifier>s.

```
<identifier> ::= <identifier> a->z  |
                 a->z  !
```

In this case, if the resulting parser is in the process of recognizing an <identifier>, then any character not satisfying the specification, a->z (i.e. any character that is not a small letter) causes the recognition of the <identifier> to terminate.

A.3.4 Syntax Specification - Conventions and Restrictions

In the following discussion, we describe the conventions and restrictions that are implemented on LIS with regard to syntax specifications. Some of these have already been discussed, and in such cases, the present discussion summarizes and extends the previous one. Most of the restrictions that have been implemented have to do with characteristics of well structured and logically complete language specifications, in general, and are thus quite independent of LIS.

All error messages delivered by LIS are directed to the user input/output stream, normally the terminal. Many of the procedures that verify the following conventions and restrictions include in their messages, an identification of the BNF rule number(R) and alternative number(A) in the form, (R: A). In tracking down the rule in question, the user may, of course, manually count the rules in the LIS Language Definition. Perhaps more simply, he may refer to the Semantic Source Segment computed during the activation, in which each rule has been placed in a PL/I comment and preceded by the label, bnf_rule(R), where R is the rule number (see example in Section A.7).

__<primary_non_terminal>__

<primary_non_terminal> is reserved on LIS, and its definition serves to identify the goal symbol of the primary grammar. Restrictions on its use are as follows:

a. If LIS is activated for purposes of computing both a primary and a lexical parser, or if it is activated to compute only a primary parser, then <primary_non_terminal> must be defined.

b. <primary_non_terminal> must __not__ be refe) iced as a symbol in an alternative.

There is no fundamental restriction on the definition of <primary_non_terminal>.

__<lexical_non_terminal>__

<lexical_non_terminal> is reserved on LIS, and its definition serves to identify the goal symbol of the lexical grammar. Restrictions on its use are as follows:

a. If LIS is activated for purposes of computing both a primary and a lexical parser, or if it is activated to compute only a lexical parser, or if it is activated to compute only a primary parser and the primary grammar has references to <lexical_non_terminal>, then <lexical_non_terminal> must be defined.

b. <lexical_non_terminal> may only be defined in BNF rules whose alternatives consist of single non-terminal symbols.

c. <lexical_non_terminal> must not be referenced as a symbol in an alternative.

d. In those cases in which LIS is used to compute both a primary parser and a lexical parser, but in __separate__ activations, using __separate__

LIS Language Definitions, the system requires that the rules defining <lexical_non_terminal> be the same in each definition, and that they appear (in the same order) as the first rules in each definition.

<non_lexical>

<non_lexical> is reserved on LIS, and its definition serves to identify those characters that are to be the explicit delimiters and spacing characters of the primary grammar. Restrictions on its use are as follows:

a. <non_lexical> should be defined with the lexical grammar in those cases in which the primary and lexical grammars are in separate LIS Language Definitions.

b. <non_lexical> may only be defined to consist of a maximum of eight single characters, i.e. a maximum of eight alternatives, each consisting of a single symbol, a character.

c. <non-lexical> must not be referenced.

d. If not defined, <non_lexical> assumes the default values of blank, tab, new-line, and new-page characters.

<any_string>

LIS has reserved the non-terminal, <any_string> so as to admit the convenient and efficient representation of language constructs such as quoted strings. The use of <any_string> is restricted so that:

a. <any_string> must not be defined.

b. An alternative in which <any_string> is referenced must consist of exactly three symbols, the first and third of which are terminal character strings, and the second of which is <any_string>.

The interpretation to be associated with the use of <any_string> is that the resulting parser, upon detecting the first terminal character string, will accept all characters up to an occurrence of the last terminal string as belonging to <any_string> for that construct.

An example of the use of <any_string> is the following definition of <quoted_string>:

<quoted_string> ::= "<any_string>" ;

Escaping Conventions

An escaping convention has been established on LIS so as to permit an alternate representation of ASCII characters. The escaping convention may be applied to any character in any rule, although when applied to character strings that are key to the LIS version of BNF ("<", ">", "::=", ";", and "!") the convention is that these symbols lose their key status. The escaping character is the apostrophe ("'"), and it may be followed either by the ASCII graphic representation of the character (e.g. '$) or by the Multics ASCII code of the character (e.g. '044). To escape the apostrophe, a double apostrophe is used.

References and Definitions of Non-Terminals

All non-terminals that are referenced must also be defined, with the following exceptions:

a. <any_string> must not be defined.

b. If LIS is activated for purposes of computing only a primary parser, then those non-terminals that are <lexical_non_terminal>s need not (although they may) be defined.

Un-needed Productions

So as to aid the LIS user in debugging the syntax specification of his language, the system has the capability to detect and report rules and/or alternatives that are not needed, i.e., that do not contribute constructs of the language. Un-needed productions are identified as a result of either structural repetition or structural gaps within the grammar, and often these are traced to simple spelling errors in the specification. Since it is sometimes the case that the user expects un-needed productions in his language, the detection of such productions will not prevent the system from attempting to compute the appropriate parsers. Such a case of expected un-needed productions may occur, for example, when a language is being developed in parts, and those parts contain definitions in common. When certain of these parts are subsequently merged to synthesize a more comprehensive subset of the language, un-needed productions may result from duplicate definitions.

In the following discussion, we present the conditions under which un-needed rules and/or alternatives may arise.

a. Duplicate Productions
In those cases in which duplicate productions exist, only the first is used in the definition, and the remaining productions are identified as not contributing to the language.

b. Productions of the Form: <A> ::= <A> Productions of this form clearly do not contribute new constructs to the language, in fact, they give rise to syntactic ambiguity. All occurrences of such productions are indicated as not contributing to the language.

c. Non-Terminals not Referenced by the Grammar
In this case, we determine which non-terminals are referenced by the grammar as follows:

   i. The set of non_terminals referenced by the grammar is initialized with the appropriate grammar goal symbol (<primary_non_terminal> or <lexical_non_terminal>).

   ii. The closure of the set is obtained by recurring on the condition that any non-terminals referenced in definitions of non-terminals that are referenced by the grammar are also referenced by the grammar.

Any non-terminals not identified as being referenced by the grammar are un-needed, and all productions which define or reference these non-terminals are identified as not contributing to the language.

d. Non-terminals not occurring in any Derivation of Text of the Language
Satisfying the referenced condition in point c above insures that the non-terminals in question are referenced by the grammar. This does not indicate, however, whether the non-terminals actually participate in the derivation of text of the language. In other words, point c identifies those non-terminals that may be found in

sentential forms of the language whereas our present concern is with those non-terminals that may occur in the derivation of _terminal_ sentential forms. Those non-terminals that do not occur in the derivation of some terminal sentential form are un-needed, and productions defining or referencing these non-terminals are identified as not contributing to the language.

Primary and Lexical Non-Terminal Conflicts

It is an implementation restriction on LIS that the same non-terminals may not be explicitly referenced by both the primary and lexical grammars, with the following exceptions:

a. <any_string> may be explicitly referenced by both grammars.

b. Those non-terminals that constitute the <lexical_non_terminal>s may be explicitly referenced by both grammars.

The CLR(k) Condition

Given that each of the relevant conditions above is satisfied, LIS will attempt to compute a parser from the appropriate syntax specification. The successful completion of this computation is dependent upon the language in question being CLR(k) for k less than or equal to three. Recall that the CLR(k) condition implies that the syntactic identity of a particular construct may be ascertained by looking indefinitely far to the left and at most k symols to the right of the current position in the parse. Failure to satisfy this condition implies the existence of a local syntactic ambiguity that cannot be resolved with up to k symbols of look-ahead. The implementation has restricted k to three because of the inefficiency incurred in parsing languages requiring more than a limited amount of look-ahead, and because k = 3 is likely to cover virtually all artificial languages of "practical" interest.

Of course, in the process of developing a language on LIS, it is likely that the user will (hopefully inadvertently) define certain constructs that are not CLR(k), $k \leq 3$ and in these cases the system delivers diagnostics describing the areas of local ambiguity.

It should be realized that the failure of a particular construct to satisfy the CLR(k) condition is no particular adverse reflection on the condition, since such constructs represent areas of complexity that are likely to be difficult to handle under any parsing scheme. Perhaps more importantly, they represent complexities in the language that users of the language would probably rather avoid.

A.4 The Definition of Artificial Languages - Specification of Semantics

In this section we discuss the way in which the language designer/implementer formulates the specification of the semantics of his language for processing by LIS. Semantic specification on LIS consists of initialization semantics and BNF rule semantics, all expressed in PL/I and employing the full capability of the language that is appropriate at the particular point. Conclusion or wrap-up semantics is a special case of rule semantics, being associated with the last production that is applied during recognition of legal text of the language. The semantic activation sequence employed in LIS computed language processors is for the initialization semantics to be activated prior to the start of the parse, and thereafter, for conrol to be passed to the semantics associated with a particular BNF rule when a construct specified by an alternative of that rule is recognized in the Input Text.

With respect to the specification of rule semantics, it is clear that the semantics to be performed is dependent on the alternative being applied. Therefore, provision has been made so that the alternative being applied is identified when control is passed to the rule semantics.

The overall structure of an LIS Language Definition is indicated in Figure A.2. The LIS Pre-Processor transforms the definition into a Semantic Source Segment, and in so doing, establishes the semantic linkages between the rules and their semantics that are necessary at language processing time. This transformation involves the generation of text, which is discussed below and which is indicated, by example, in Section A.7.

A.4.1 Initialization Semantics

Initialization semantics exists so as to permit the language designer/implementer to specify those semantic actions to be performed by his processor prior to the initiation of the parsers. Initialization semantics is, by convention, all text prior to the first BNF rule.

It is not necessary to include a PL/I procedure statement at the beginning of the initialization semantics, since this is done automatically by LIS when computing the Semantic Source Segment from the LIS Language Definition. The label on the procedure statement is derived from the segment name of the language definition, so that if the segment is named "abc.lis", the label on the procedure statement is "abc" (for information on LIS segment naming conventions, see Section A.6). Just prior to parsing, the

```
                    .
                    .
          Initialization Semantics
                    .
                    .
                    .
                    .
                    .
BNF Rule    --------------------
                    .
                    .
          Rule Semantics (↑)
                    .
                    .
                    .
BNF Rule    --------------------
                    .
                    .
          Rule Semantics (↑)
                    .
                    .
                    .
BNF Rule    --------------------
                    .
                    .
          Rule Semantics (↑)
                    .
                    .
                    .
BNF Rule    --------------------
                    .
                    .
          Rule Semantics (↑)
                    .
                    .
                    .
BNF Rule    --------------------
                    .
                    .
          Rule Semantics (↑)
                    .
                    .
                    .
```

<u>LIS Language Definition Structure</u>

Figure A.2

initialization semantics is activated via a call to abc_semantics$abc made by LIS Processor Control. It is the responsibility of the initialization semantics to return control to Processor Control (via the PL/I return statement) upon completion of its actions.

A.4.2 BNF Rule Semantics

The semantic interpretation to be associated with a particular BNF rule is understood to be the complete (PL/I) text between that rule and the next rule, or end of segment if the rule in question is the last rule of the definition. If the text consists of blank, tab, new-line, and new-page characters *exclusively*, then it is assumed that there is *no* semantic interpretation to be associated with the rule. Therefore, during language processing, recognition of the associated syntactic constructs *will not* result in a transfer of control to the semantic segment by LIS Processor Control. Any text other than blank, tab, new_line, or new_page characters *will* be interpreted as significant semantic text, and will result in the transfer of control upon recognition of the associated constructs.

The following transformations on the LIS Language Definition are performed automatically by LIS, and establish the necessary semantic linkage with the syntax specification:

a. The following text is inserted between the user defined initialization semantics and the first BNF rule:

```
 bnf_rule_semantics: entry(bnf_rule_number,
                     alternative_number);

 dcl  alternative_number  fixed   binary(35),
      bnf_rule(1000)      label,
      bnf_rule_number     fixed   binary(35);

 go to bnf_rule(bnf_rule_number);
```

alternative_number is set during calls to the semantic procedure, and indicates which of the possible alternatives of a rule is being applied at the particular point during the parse.

b. Each BNF rule is placed in a PL/I comment and preceded by the label "bnf_rule(n)", where n is the sequence number of the rule in the definition (n = 1 for the first rule, n = 2 for the second rule, etc).

The way in which linkage is effected to the BNF rule semantics may now be explained:

Assume an LIS produced processor is processing text written in the language specified by the definition in segment abc.lis. When a reduction is to be made by alternative q of BNF rule p, the following call is made:

```
     call abc_semantics$bnf_rule_semantics(p, q);
```

This results in control being passed to the rule semantics associated with the p-th rule, via the statement:

```
     go to  bnf_rule(bnf_rule_number);
```

where bnf_rule_number has the value p. It is the responsibility of the rule semantics subsequently to return control to Processor Control (via the PL/I return statement).

A.4.3 References to Input Text

The language semantics gains access to the input text by including the following declarations:

```
 1     text_reference_stack(100)      aligned     external,
       2    construct_start           fixed       binary(35),
       2    construct_length          fixed       binary(35),
top                   fixed           binary(35)  external,

 1     input_text_struct              aligned
                           based(input_text_struct_ptr),
       2    input_text                char(input_text_length),
 input_text_struct_ptr                pointer     external,
 input_text_length         fixed      binary(35)  external,
```

References to the input text are made through the based character string, input_text. These accesses are normally coordinated, however, by making use of the run-time text reference stack, text_reference_stack.

The general rules by which access to the input text is coordinated by the text reference stack may be stated as follows:

a. Consider a particular alternative of a BNF rule for which semantics is to be specified. This alternative has "n" symbols:

   symbol(1) symbol(2)....symbol(i)....symbol(n)

b. During parsing, when a reduction is to be made according to the alternative in question, the top "n" entries on the text reference stack refer to the symbols of the alternatives as follows:

```
text_reference_stack(top-n+1) refers to symbol(1)
text_reference_stack(top-n+2) refers to symbol(2)
                                .
                                .
text_reference_stack(top-n+i) refers to symbol(i)
                                .
                                .
text_reference_stack(top)     refers to symbol(n)
```

"top" is set by LIS Processor Control just prior to transferring control to the semantics associated with the rule in question.

c. The elements of text_reference_stack, construct_start and construct_length, refer, respectively, to the starting character position and the length, in characters, of the <u>first</u> lexical construct recognized as part of the referenced symbol.

d. The first lexical construct of the i-th symbol of the alternative may then be accessed by way of the PL/I built-in function, substr, as follows:

```
substr(input_text, construct_start(top-n+i),
        construct_length(top-n+i))
```

The following two examples illustrate the way in which the text reference stack may be used to coordinate references to the input text.

<u>Example 1</u>

A stack entry associated with a symbol of an alternative that is a lexical construct refers to the occurrence of that construct in the input text. For example, suppose that semantics is to be specified for the following rule (we assume here that <integer> is a <lexical_non_terminal>):

```
<subscript> ::= ( <integer> ) !
```

The stack entries that are relevant for accessing the symbols associated with this rule are as follows:

```
text_reference_stack(top)      refers to ")"
text_reference_stack(top-1)    refers to <integer>
text_reference_stack(top-2)    refers to "("
```

In recognizing "(3)" as a <subscript> the application of substr could be made (using the "top-1"th entry of text_reference_stack) to gain access to the character, "3". (Of course, it would not be necessary to fetch the symbols "(" and ")" in this manner, since application of the given production implies that stack entries top and top-2 will, a priori, refer to ")" and "(", respectively.

Example 2

The stack entry associated with a symbol of an alternative that is not a lexical construct does not refer to the entire symbol, only to the first lexical construct recognized as part of the symbol. Consider, for example, the following two rules, with the associated text references as indicated (we assume here that <identifier> is a <lexical_non_terminal>):

```
Rule 1
<assignment> ::= <identifier> = <expression> ; !

text_reference_stack(top)      refers to ";"
text_reference_stack(top-1)    refers to first lexical
                                         construct of
                                         <expression>
text_reference_stack(top-2)    refers to "="
text_reference_stack(top-3)    refers to <identifier>


Rule 2
<expression> ::= <expression> + <identifier> !
                 <identifier> !

Alternative 1
text_ reference_stack(top)     refers to <identifier>
text_reference_stack(top-1)    refers to "+"
text_reference_stack(top-2)    refers to first lexical
                                         construct of
                                         <expression>
```

Alternative 2
text_reference_stack(top)   refers to <identifier>

Parsing of "a = b + c;" then proceeds as follows:

a. The <identifier>, "b" would be recognized as an <expression> (Rule 2, Alternative 2). At this point, text_reference_stack(top) would refer to "b";

b. "+" and "c" would be recognized, and a reduction according to Rule 2, Alternative 1 would be performed. The stack entries would then be as follows:

```
text_reference_stack(top)   refers to "c"
text_reference_stack(top-1) refers to "+"
text_reference_stack(top-2) refers to "b"
```

c. The <identifier>, "a" and the equal sign, "=", having already been recognized, ";" would be recognized, and a reduction according to Rule 1 would be performed. The stack entries would then be as follows:

```
text_reference_stack(top)   refers to ";"
text_reference_stack(top-1) refers to "b"
text_reference_stack(top-2) refers to "="
text_reference_stack(top-3) refers to "a"
```

Further examples on the referencing of lexical constructs during the LIS CLR(k) parse may be found in Section A.7.

The correspondence of the text rererence stack entries to the lexical constructs is part of the overall design of LIS and is therefore not easily modified. However, the way in which references are made to the actual text of the constructs is quite flexible and involves only minor modifications to LIS Processor Control. In the above discussions, these references were made by way of the PL/I substr built-in function in conjunction with input_text, construct_start, and construct_length. An alternative referencing strategy is to enter the text of the constructs into the stack directly, thus alleviating the rather formidable substr expressions. Note also, that this strategy removes the requirement for maintaining the entire Input Text during the parse. Of course, there are tradeoffs involving the increased space requirements for the text reference stack versus the convenience of direct reference, and these tradeoffs will have to be evaluated for each particular situation.

A.5 LIS Processor Control

LIS Processor Control is the procedure responsible for coordinating the language processing activity. In parsing Input Text, it is driven by the DPDAs. With respect to language semantics, its task is to coordinate the parse with the user defined semantics so that the appropriate semantics may be activated upon recognition of the corresponding construct.

The configuration of LIS Processor Control for a particular language will be provided and maintained by the LIS System Maintainance Group.

A.6 Using the Multics Implementation of LIS

The following pages describe the mechanisms by which the language designer/implementer utilizes LIS on Multics for purposes of developing artificial languages and their associated processors.

Command
In Directory: >udd>LIS>Altmanv>LIS_SYSTEM
Vernon E. Altman

Name: lis

The lis command invokes the Language Implementation System to process an LIS Language Definition. The functional output of LIS consists of a set of tables (DPDAs) and a PL/I procedure, which are combined with LIS Processor Control to synthesize a processor for the language being defined.

Processing a Language Definition

The command:

```
lis <path-name> [<options>...]
```

invokes LIS to process the definition segment specified by <path-name>. A directory pathname, <directory-name> and an entry name, <segment-name> are derived from <path-name>. Due to the Multics restriction on segment name lengths, the length of <segment-name> is limited to 18 characters.

Options

The <option>s with which LIS may be invoked are as follows:

(pl:p:l) One of these options may be specified, with the following meaning:

pl LIS will compute DPDAs from the primary grammar and the lexical grammar. In addition, the Semantic Source Segment will be computed. This is the default option.

p LIS will compute the primary grammar DPDA. In addition, the Semantic Source Segment will be computed.

l LIS will compute the lexical grammar DPDA. In addition, the Semantic Source Segment will be computed.

t If this option is specified, LIS will print out the timing statistics of the major system modules as processing progresses. The timing results are unlikely to be of direct interest to the general user. However, they do give an accurate indication of the of processing sequence and of the total processing cost. The default for this option is not to display timing statistics.

sem If this option is specified, LIS will compute the semantic Source Segment only. This option overrides any of the options (pl:p:l). The default for this option is off.

If LIS is invoked with no arguments (i.e., simply "lis"), then it will display on the user input/output stream (normally the terminal) the form in which it expects its arguments.

Generated Segments

LIS generates the following segments in the user's working directory, depending on the options and validity of the definition as indicated.

<segment-name>.lis_exec

> This segment contains printable information on the processing of the definition that should prove to be of value to the user in developing his language and its processor. In the current implementation of LIS, this segment receives the non-terminal cross-references (definitions and references) and a listing of the key-symbols of the language being processed. This segment is generated in all activations of LIS except those in which the input segment is syntactically invalid (e.g. LIS is activated to process a PL/I object segment), or in which the "sem" option is specified.

<segment-name>_semantics.pl1

> This is the Semantic Source Segment that LIS computes from the LIS Language Definition. This segment is always computed in those activations of LIS in which an LIS Language Definition segment is specified.

<segment-name>.dpda

> This segment contains the DPDAs (parsing tables) that are computed from the LIS Language Definition, and that are used to drive LIS Processor Control in the the parse of Input Text written in the defined language. This segment is computed if and only if the submitted grammar is found to be acceptable, both to the LIS Pre-Processor and to the LIS CLR(k) Generator, and if the "sem" option is not specified.

<segment-name>.lis_dpda

This segment is simply a printable version of <segment-name>.dpda.

Processor Control

The configuration of LIS Processor Control for a particular language will be provided and maintained by the LIS System Maintenance Group.

A.7 Example of an LIS Language Definition

In this section, we present an example of the utilization of LIS in the development of an artificial language and its associated processor. The language defines a simple arithmetic assignment statement, called assign. The syntax of assign is as follows:

The Lexical Grammar

```
<lexical_non_terminal> ::=
                         <identifier>  ;
                         <integer>  !

<identifier> ::=
                         <identifier> a->z  ;
                         a->z  !

<integer> ::=
                         <integer> 0->9  ;
                         0->9  !

<non_lexical> ::=
                         '040 ; '011 ; '012 ; '014  !
```

The Primary Grammar

```
<primary_non_terminal> ::=
                         <assignment_statement>  !

<assignment_statement> ::=
                         <identifier> = <expression> ;  !

<expression> ::=
                         <expression> + <term>  ;
                         <expression> - <term>  ;
                         <term>  !

<term> ::=
                         <term> * <factor>  ;
                         <term> / <factor>  ;
                         <factor>  !
```

```
<factor> ::=
                        ( <expression> )  |
                        <identifier>  |
                        <integer>  |
```

The processing that we perform on text of the assign language is a simple translation into a symbolic intermediate language, the type that may be produced by the interpretation phase of a typical compiler, for example. Admittedly, the example is far too simple to be of any practical value: its purpose is simply to illustrate how the syntax and semantics of artificial languages may be structured for processing by LIS. As such, the example draws on many of the language design and processor implementaion issues previously discussed.

Processor Control for our translator is an "off the shelf" version, and is named assign. assign takes its input from segments with the suffix, ".assign".

On the following pages, we have included those documents associated with assign that are particularly relevant to the LIS user during development of the language. The documents are:

1. The LIS Language Definition.
2. The Semantic Source Segment.
3. The LIS Execution Output Segment.
4. Sample Translations.

The semantics of the language is reasonably straightforward and self-documenting. Our discussion of the Language Definition will therefore be limited to the following description of the output language primitives:

```
add operand_1, operand_2, operand_3:
     operand_1 + operand_2 -> operand_3

sub operand_1, operand_2, operand_3:
     operand_1 - operand_2 -> operand_3

mult operand_1, operand_2, operand_3:
     operand_1 * operand_2 -> operand_3

div operand_1, operand_2, operand_3:
     operand_1 / operand_2 -> operand_3

assign operand_1, operand_2:
     operand_1 -> operand_2
```

The translation makes use of temporary variables, which are indicated in the output as: T<integer>.

```
          assign.lis          05/01/73  0335.8 edt Tue


/*      Declarations and Initialization Semantics        */

        dcl     1       text_raferance_stack(100)   aligned   external  static,
                        2       construct_start     fixed     binary(35),
                        2       construct_length    fixed     binary(35),
                top                                 fixed     bin(35)   external  static,

                1       input_taxt_struct           aligned   based(input_text_struct_ptr),
                        2       input_text          char(input_text_length),
                input_text_struct_ptr               pointer   external  static,
                input_text_length                   fixed     bin(35)   external  static,


                expression_stack(100)        char(10)  unaligned,
                expression_stack_top         fixed     binary(35),
                fb_to_char                   entry(fixed binary(17))      internal
                                             returns(char(13) varying),
                ioa_                         entry     external,
                new_temp                     entry     internal  returns(char(10) unaligned),
                number_of_temps              fixed     binary(17),
                opcode                       char(4)   unaligned,
                operand_1                    char(10)  unaligned,
                operand_2                    char(10)  unaligned,
                operand_3                    char(10)  unaligned,
                pop                          entry     internal  returns(char(10) unaligned),
                push                         entry(char(10) unaligned)     internal;


        expression_stack_top = 0;
        number_of_temps = 0;
        return;


<lexical_non_terminal> ::=
                    <identifier>  |
                    <integer>  !

<identifier> ::=
                    <identifier> a->z  |
                    <identifier> 0->9  |
                    <identifier> _  |
                    a->z  !

<integer> ::=
                    <integer> 0->9  |
                    0->9  !

<non_lexical> ::=
                    ^040 | ^011 | ^012 | ^014  !
```

```
/*        The Primary Grammar        */

<primary_non_terminal> ::=
                    <assignment_statement>  !

          call ioa_("^/Assignment Translation Completad");
          return;


<assignmant_statemant> ::=
                    <identifier> = <expression> ;  !

          operand_1 = pop;
          operand_2 = substr(input_taxt, construct_start(top - 3), construct_length(top - 3));
          call ioa_("^-assign ^a, ^a", operand_1, operand_2);
          return;


<expression> ::=
                    <axpression> + <tarm>  :
                    <axpression> - <term>  :
                    <term>  !

          if alternative_number = 3
             than return;
          if alternative_number = 1
             then opcoda = "add";
             else opcode = "sub";

standard_expression_evaluation:
          operand_2 = pop;
          operand_3 = new_temp;
          operand_1 = pop;
          call ioa_("^-^a ^a, ^a, ^a", opcode, operand_1, operand_2, oparand_3);
          call push(operand_3);
          return;


<term> ::=
                    <term> * <factor>  :
                    <term> / <factor>  :
                    <factor>   !

          if alternative_number = 3
             then return;
          if alternative_number = 1
             than opcode = "mult";
             else opcode = "div";
          go to standard_expression_evaluation;
```

```
<factor> ::=
                    ( <expression> )  :
                    <identifier>  :
                    <integer>  ;

          if alternative_number = 1
             then return;
          operand_1 = substr(input_text, construct_start(top), construct_length(top));
          call push(operand_1);
          return;



new_temp:   proc  returns(char(10) unaligned);   /*                                    *     new_temp     *
                                                                                       ********************
          This procedure creates a new temporary variable.  */

          dcl       temp                char(10)  unaligned;


          number_of_temps = number_of_temps + 1;
          temp = "T"::fb_to_char(number_of_temps);
          return(temp);
          end;



push:     proc(top_of_stack);   /*                                                     *     push     *
                                                                                       ****************
          This procedure pushes the variable, top_of_stack, onto the
          exoression stack.  */

          dcl       top_of_stack        char(10)  unaligned;


          exoression_stack_too = expression_stack_top + 1;
          expression_stack(expression_stack_top) = top_of_stack;
          return;
          end;



pop:      proc  returns(char(10) unaligned);   /*                                      *     pop     *
                                                                                       ***************
          This procedure pops the top variable off the expression stack and
          returns it.  */

          dcl       top_of_stack        char(10)  unaligned;


          top_of_stack = expression_stack(expression_stack_top);
          expression_stack_top = expression_stack_top - 1;
          return(top_of_stack);
          end;
```

```
                                                                              *      fb_to_char      *
fb_to_char;   proc(v)  returns(char(13) var);   /*                            ************************
        This procedure converts a fixed binary number into its
        equivalent character string representation.  */

        dcl                   a         fixed     bin(35),
                              c         char(4)   based(addr(m)),
                              ii        fixed     bin(35)   init(14),
                              m         fixed     bin(35),
                              neg       bit(1),
                              r         char(13)  varying,
                              s         char(13),
                              v         fixed     bin(35);


        neg = "0"b;
        if v = 0  then return("0");
        if v < 0  then neg = "1"b;
        a = abs(v);
        do while(a>0);
                m = mod(a, 10) + 48;
                ii = ii - 1;
                substr(s, ii, 1) = substr(c, 4, 1);
                a = divide(a, 10, 35, 0);
        end;
        r = substr(s, ii);
        if neg  then r = "-"||r;
        return(r);
        end;
```

```
                assign_semantics.pl!          05/ 1/7.  0405.6 edt Tue

assign_semantics:   proc;
                                                          /*     assign_semantics     */

*       Declarations and Initialization Semantics        */

        dcl        1          text_reference_stack(100)       aligned    external  static,
                              2          construct_start      fixed      binary(35),
                              2          construct_length     fixed      binary(35),
                   top                                        fixed      bin(35)    external  static,

                   1          input_text_struct               aligned    based(input_text_struct_ptr),
                              2          input_text           char(input_text_length),
                   input_text_struct_ptr                      pointer    external  static,
                   input_text_length                          fixed      bin(35)    external  static,


                   expression_stack(100)        char(10)  unaligned,
                   expression_stack_top         fixed     binary(35),
                   fb_to_char                   entry(fixed binary(17))       internal
                                                returns(char(13) varying),
                   ioa_                         entry     external,
                   new_temp                     entry     internal  returns(char(10) unaligned),
                   number_of_temps              fixed     binary(17),
                   opcode                       char(4)   unaligned,
                   operand_1                    char(10)  unaligned,
                   operand_2                    char(10)  unaligned,
                   operand_3                    char(10)  unaligned,
                   pop                          entry     internal  returns(char(10) unaligned),
                   push                         entry(char(10) unaligned)     internal;


        expression_stack_top = 0;
        number_of_temps = 0;
        return;



bnf_rule_semantics:   entry(bnf_rule_number, alternative_number);   /*        *     bnf_rule_semantics     *
                                                                              *******************************

        This entry affects the semantic linkage to the BNF rule semantics.   */

        dcl        alternative_number          fixed     binary(35),
                   bnf_rule(1000)              label,
                   bnf_rule_number             fixed     bin(35);

        go to bnf_rule(bnf_rule_number);
```

```
bnf_rule(1):        /* <lexical_non_terminal> ::=
                    <identifier>  |
                    <integer>  ! */


bnf_rule(2):        /* <identifier> ::=
                    <identifier> a->z  |
                    <identifier> 0->9  |
                    <identifier> _  |
                    a->z  ! */


bnf_rule(3):        /* <integer> ::=
                    <integer> 0->9  |
                    0->9  ! */


bnf_rule(4):        /* <non_lexical> ::=
                    ^040 | ^011 | ^012 | ^014  ! */




/*      The Primary Grammar       */


bnf_rule(5):        /* <primary_non_terminal> ::=
                    <assignment_statement>  ! */

          call ioa_("^/Assignment Translation Completed");
          return;


bnf_rule(6):        /* <assignment_statement> ::=
                    <identifier> = <expression> ;  ! */

          operand_1 = pop;
          operand_2 = substr(input_text, construct_start(top - 3), construct_length(top - 3));
          call ioa_("^-assign ^a, ^a", operand_1, operand_2);
          return;

bnf_rule(7):        /* <expression> ::=
                    <expression> + <term>  |
                    <expression> - <term>  |
                    <term>  ! */

          if alternative_number = 3
             then return;
          if alternative_number = 1
             then opcode = "add";
             else opcode = "sub";
```

```
s.ndard_expression_evaluation:
          operand_2 = pop;
          operand_3 = new_temp;
          operand_1 = pop;
          call ioa_("^-^a ^a, ^a, ^a", opcode, operand_1, operand_2, operand_3);
          call push(operand_3);
          return;


bnf_rule(8):        /* <term> ::=
                    <term> * <factor>  !
                    <term> / <factor>  !
                    <factor>   ! */


          if alternative_number = 3
             then return;
          if alternative_number = 1
             then opcode = "mult";
             else opcode = "div";
          go to standard_expression_evaluation;


bnf_rule(9):        /* <factor> ::=
                    ( <expression> )  !
                    <identifier>  !
                    <integer>  ! */


          if alternative_number = 1
             then return;
          operand_1 = substr(input_text, construct_start(top), construct_length(top));
          call push(operand_1);
          return;



new_temp:   proc  returns(char(10) unaligned);   /*                           *     new_temp     *
                                                                              ********************
          This procedure creates a new temporary variable.  */

          dcl       temp                char(10)  unaligned;


          number_of_temps = number_of_temps + 1;
          temp = "T"||fb_to_char(number_of_temps);
          return(temp);
          end;
```

```
push;     proc(top_of_  ack);   /*                                        *      push      *
          This procedure pushes the variable, top_of_stack, onto the      ****************
          expression steck.  */

          dcl       top_of_stack        char(10)  unaligned;


          expression_steck_top = expression_stack_top + 1;
          expression_stack(expression_stack_top) = top_of_stack;
          return;
          end;


pop:      proc  returns(cher(10) uneligned);   /*                         *      pop       *
                                                                          ****************
          This procedure pops the top verieble off the expression stack and
          returns it.  */

          dcl       top_of_stack        char(10)  uneligned;


          top_of_stack = expression_steck(expression_stack_top);
          expression_stack_top = expression_stack_top - 1;
          return(top_of_stack);
          end;


fb_to_cher;   proc(v)  returns(char(13) var);   /*                        *     fb_to_char     *
                                                                          ********************
          This procedure converts a fixed binary number into its
          equivalent character string representetion.  */

          dcl                 a         fixed     bin(35),
                              c         char(4)   based(eddr(m)),
                              ii        fixed     bin(35)   init(14),
                              m         fixed     bin(35),
                              neg       bit(1),
                              r         cher(13)  varying,
                              s         char(13),
                              v         fixed     bin(35);


          neg = "0"b;
          if v = 0  then return("0");
          if v < 0  then neg = "1"b;
          e = abs(v);
          do while(a>0);
                    m = mod(a, 10) + 48;
                    ii = ii - 1;
                    substr(s, ii, 1) = substr(c, 4, 1);
                    a = divide(a, 10, 35, 0);
          end;
```

```
                  r = substr(s, i1);
                  if neg  then r = '-'||r;
                  return(r);
                  end;
end               assign_semantics;
```

```
          assign.lis_exec     05/01/73  0422.8 edt Tue

          The  Non-Terminal  Cross-References  For:  >udd>LIS>Altmanv>LIS_DOCUMENTATION>assign.lis


<primary_non_terminal>               Defined:     5
                                     Referenced: No References

<lexical_non_terminal>               Defined:     1
                                     Referenced: No References

<non_lexical>                        Defined:     4
                                     Referenced: No References

<identifier>                         Defined:     2
                                     Referenced: (1: 1), (2: 1), (2: 2), (2: 3), (6: 1), (9: 2)

<integer>                            Defined:     3
                                     Referenced: (1: 2), (3: 1), (9: 3)

<assignment_statement>               Defined:     6
                                     Referenced: (5: 1)

<expression>                         Defined:     7
                                     Referenced: (6: 1), (7: 1), (7: 2), (9: 1)

<term>                               Defined:     8
                                     Referenced: (7: 1), (7: 2), (7: 3), (8: 1), (8: 2)

<factor>                             Defined:     9
                                     Referenced: (8: 1), (8: 2), (8: 3)




     The  Key-Symbol  Table:

     1         =
     2         :
     3         +
     4         -
     5         <identifier>        ( <lexical_non_terminal> )
     6         <integer> ( <lexical_non_terminal> )
     7         *
     8         /
     9         (
     10        )
```

```
assign
assign: arguments missing.
Arguments:
1:      <assign_input_text_segment_name>
2-3:    <options>
           "p"       Print parse of assign program.
           "ns"      Do not perform any translation
                     semantics.
```

```
print a1.assign 1
a = b + c;


assign a1 p
            add b, c, T1
            assign T1, a
Assignment Translation Completed


The LIS CLR(k) Parse of a1.assign:

Line        Action
1           read a
1           read =
1           read b
1           apply (9: 2)
1           apply (8: 3)
1           see +
1           apply (7: 3)
1           read +
1           read c
1           apply (9: 2)
1           apply (8: 3)
1           see ;
1           apply (7: 1)
1           read ;
1           apply (6: 1)
1           apply (5: 1)

Parse Completed
```

```
print a2.assign 1
left =
operand_1 + operand_2
-
12345 * (operand_3 + 6*operand_4) / operand_5 +
operand_6    ;


assign a2 p
            add operand_1, operand_2, T1
            mult 6, operand_4, T2
            add operand_3, T2, T3
            mult 12345, T3, T4
            div T4, operand_5, T5
            sub T1, T5, T6
            add T6, operand_6, T7
            assign T7, left
Assignment Translation Completed


The LIS CLR(k) Parse of a2.assign:

Line        Action
1           read left
1           read =
2           read operand_1
2           apply (9: 2)
2           apply (8: 3)
2           see +
2           apply (7: 3)
2           read +
2           read operand_2
2           apply (9: 2)
2           apply (8: 3)
3           see -
3           apply (7: 1)
3           read -
4           read 12345
4           apply (9: 3)
4           apply (8: 3)
4           see *
4           read *
4           read (
4           read operand_3
4           apply (9: 2)
4           apply (8: 3)
4           see +
4           apply (7: 3)
4           read +
4          read 6
4          apply (9: 3)
4          apply (8: 3)
4          see *
4          read *
4          read operand_4
4          apply (9: 2)
4          apply (8: 1)
4          see )
4          apply (7: 1)
4          read )
4          apply (9: 1)
4          apply (8: 1)
4          see /
4          read /
4          read operand_5
4          apply (9: 2)
4          apply (8: 2)
4          see +
4          apply (7: 2)
4          read +
5          read operand_6
5          apply (9: 2)
5          apply (8: 3)
5          see :
5          apply (7. .)
5          read :
5          apply (6: 1)
5          apply (5: 1)

Parse Completed
```

```
print a3.assign l
left =
operand_1 + operand_2
-
12345 * operand_3 & operand_4  +
operand_5    ;


assign a3 p
         add operand_1, operand_2, T1
         mult 12345, operand_3, T2
         sub T1, T2, T3
Syntax error on line 4, reading: "& op...".
Translation terminated.


The LIS CLR(k) Parse of a3.assign:

Line     Action
1        read left
1        read =
2        read operand_1
2        apply (9: 2)
2        apply (8: 3)
2        see +
2        apply (7: 3)
2        read +
2        read operand_2
2        apply (9: 2)
2        apply (8: 3)
3        see -
3        apply (7: 1)
3        read -
4        read 12345
4        apply (9: 3)
4        apply (8: 3)
4        see *
4        read *
4        read operand_3
4        apply (9: 2)
4        apply (8: 1)
4        see &
4        apply (7: 2)

Parse Terminated
```

Appendix B

LIS - A Macro System Description

B.1 Introduction

In this appendix, we present a macro description of the Language Implementation System. The description includes flowcharts of the major system procedures and descriptions of the major system data structures. This macro description, in conjunction with the algorithmic description in Chapter II, should provide enough information on the system design so that "persons having ordinary skill in the art to which said subject matter pertains" may reproduce the implementation of LIS.

B.2 Major System Procedures

In the following discussion, we present the design of the major procedures of the Language Implementation System. In addition, we indicate the size of each procedure by giving the number of PL/I statements (including comments) comprising the procedure, and by giving the procedure's object segment size (36 bits/word).

B.2.1 lis

When activated at its primary entry point, lis, or at the secondary entry point, debug_monitor_entry, the lis procedure controls the processing of an LIS Language Definition by the Language Implementation System. In such activations, its function is essentially that of a computational dispatcher in the sense that its own capabilities are restricted to the controlled activation of the major system modules of Processor Generation.

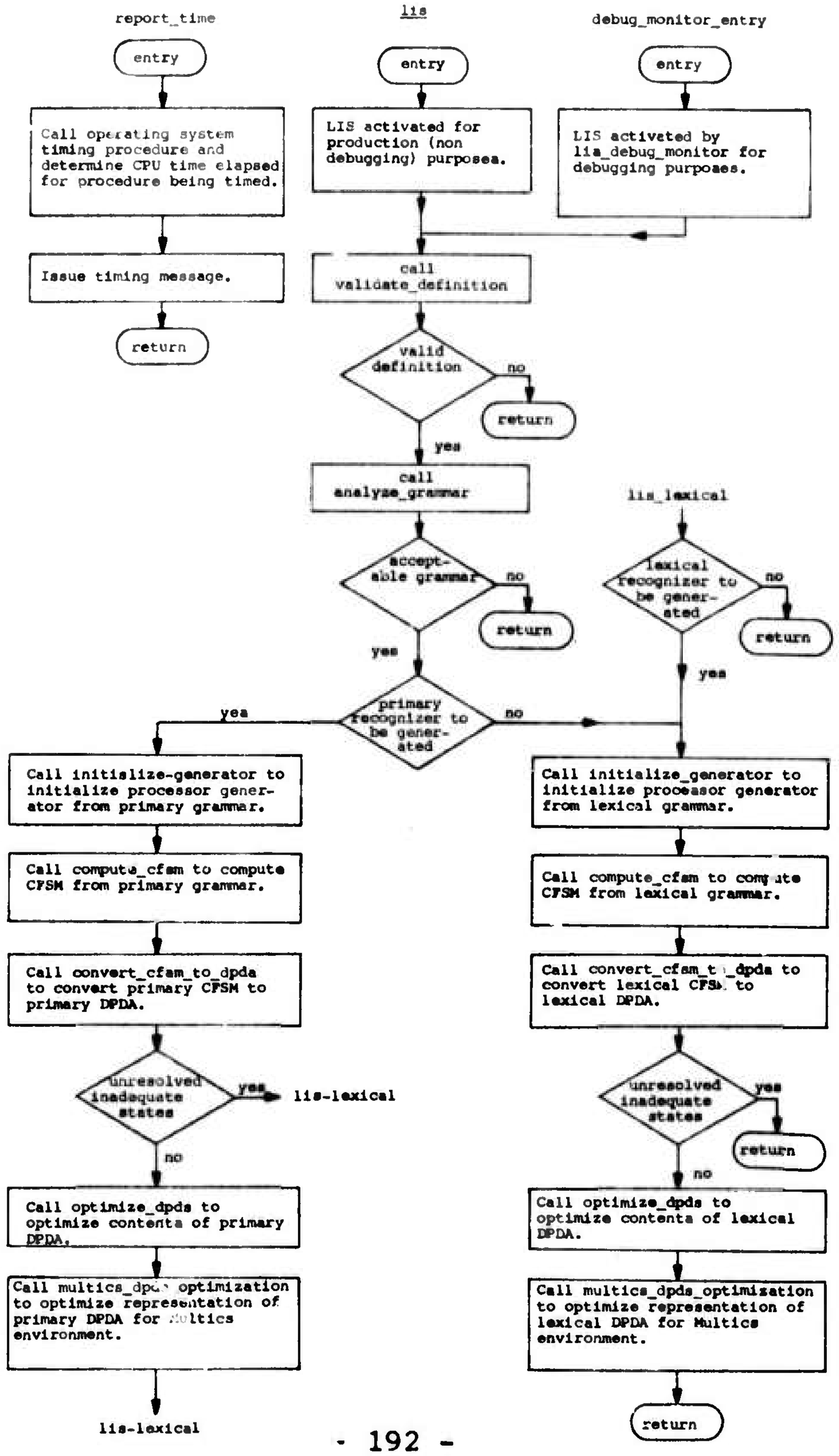
The entry points of the lis procedure are as follows:

lis
: This is the primary entry point of lis and is invoked by the language designer/implementer for purposes of processing an LIS Language Definition.

debug_monitor_entry
: This is a secondary entry point of lis and is invoked by the lis_debug_monitor procedure when modifying and debugging LIS.

report_time
: This is a secondary entry point of lis and is invoked at various points within LIS for purposes of system timing. Since it does not contribute to the functional capability of the system, there are no further references to report_time within the subsequent system description.

Procedure Size:
   Source: 168 PL/I Statements
   Object: 1137 Words



report_time
entry
Call operating system timing procedure and determine CPU time elapsed for procedure being timed.
Issue timing message.
return
lis
entry
LIS activated for production (non debugging) purposes.
debug_monitor_entry
entry
LIS activated by lis_debug_monitor for debugging purposes.
call validate_definition
valid definition
no
return
yes
call analyze_grammar
acceptable grammar
no
return
yes
primary recognizer to be generated
yes
no
lis_lexical
lexical recognizer to be generated
no
return
yes
Call initialize-generator to initialize processor generator from primary grammar.
Call compute_cfsm to compute CFSM from primary grammar.
Call convert_cfsm_to_dpda to convert primary CFSM to primary DPDA.
unresolved inadequate states
yes
lis-lexical
no
Call optimize_dpda to optimize contents of primary DPDA.
Call multics_dpda_optimization to optimize representation of primary DPDA for Multics environment.
lis-lexical
Call initialize_generator to initialize processor generator from lexical grammar.
Call compute_cfsm to compute CFSM from lexical grammar.
Call convert_cfsm_to_dpda to convert lexical CFSM to lexical DPDA.
unresolved inadequate states
yes
return
no
Call optimize_dpda to optimize contents of lexical DPDA.
Call multics_dpda_optimization to optimize representation of lexical DPDA for Multics environment.
return

B.2.2 validate_definition

The validate_definition procedure performs the following basic functions:

a. The procedure processes the arguments with which LIS was activated.

b. The procedure processes the submitted grammar by validating its syntax and entering the grammar into the grammar structures (Section B.3.1).

c. The procedure computes the Semantic Source Segment.

In performing the above functions, validate_definition makes use of the following internal procedures:

move_rule
This procedure is invoked for each BNF rule of the LIS Language Definition, and for the i-th such rule it appends to the Semantic Source Segment the PL/I label variable, bnf_rule(i), and the i-th rule enclosed in PL/I comment delimiters (e.g. /* i-th rule */). The text index (into the LIS Language Definition segment) is left pointing at the character just beyond the end of the i-th BNF rule.

fetch_non_terminal
When this procedure is invoked, the text index is pointing at a character that is suspected to be the first character of a non-terminal of the grammar (e.g. "<"). fetch_non_terminal then determines if, in fact, a non-terminal exists which starts at that point. If so, the non-terminal is entered into the non-terminal structure (this structure contains an entry for each unique non-terminal in the grammar - see Section B.3.2), the index of its entry in the structure is returned, and the text index is advanced to the end of the non-terminal (e.g. ">"). If non-terminal is not detected, then an entry index of zero is

returned and the text index is not modified.

fetch_terminal_string
When this procedure is invoked, the text index is pointing at a character that is known to be the start of a terminal string of the grammar. fetch_terminal_string then determines the length of the terminal string by scanning for the first non-escaped blank, tab, new line, or new page character. The input text index is advanced to the end of the terminal string and the starting index and length of the terminal string are passed to the calling procedure.

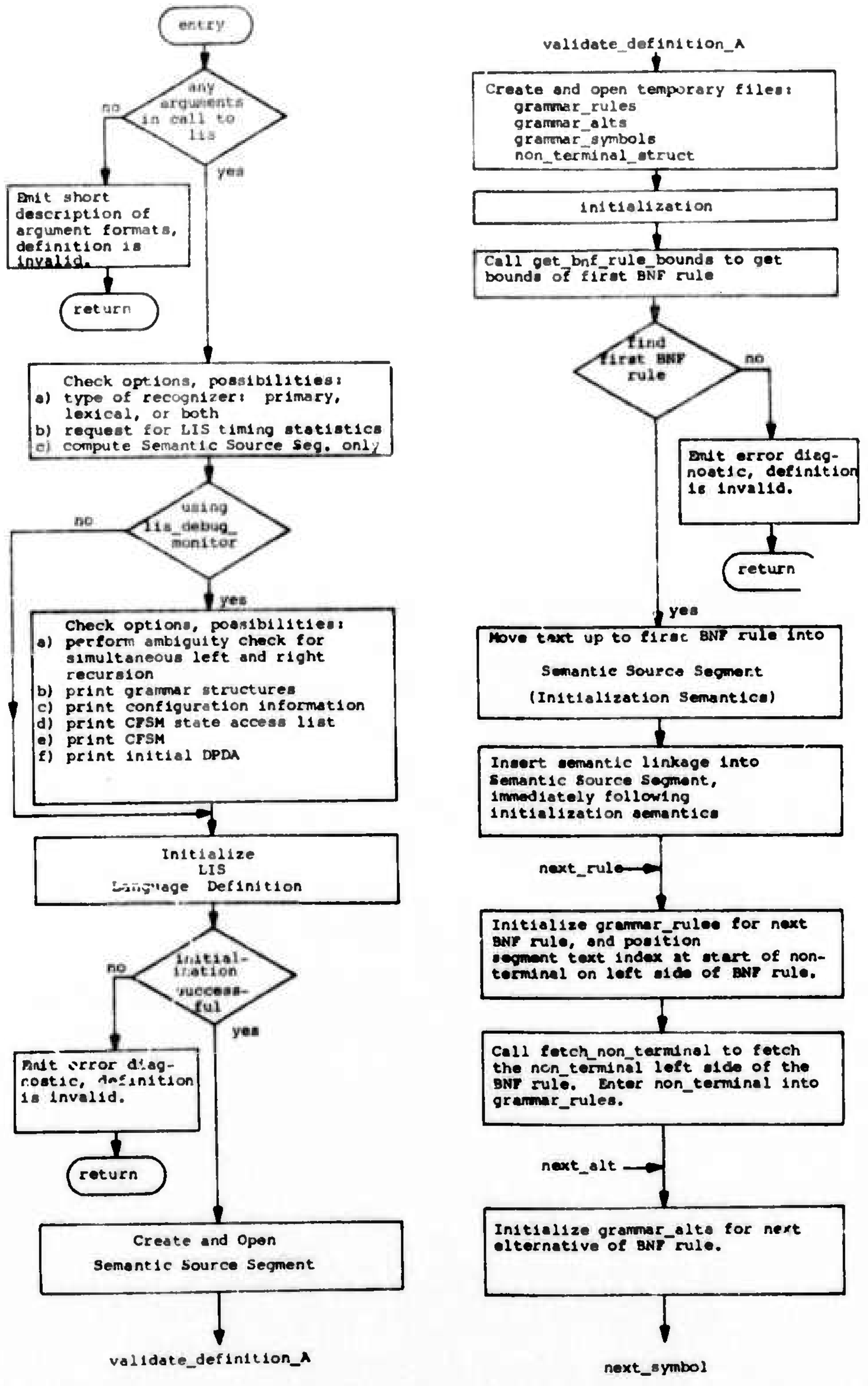
get_bnf_rule_bounds
This procedure scans the LIS Language Definition segment from the current text position, looking for the next BNF rule of the grammar. If a rule is found, the location of its starting position and ending position are passed to the calling procedure. If no rule is found, values of zero are returned for the starting and ending positions. The text index is not modified by this procedure.

get_non_space
This procedure advances the text index from its current position to a position at which it points at a character that is neither a blank, tab, new line, or new page character.

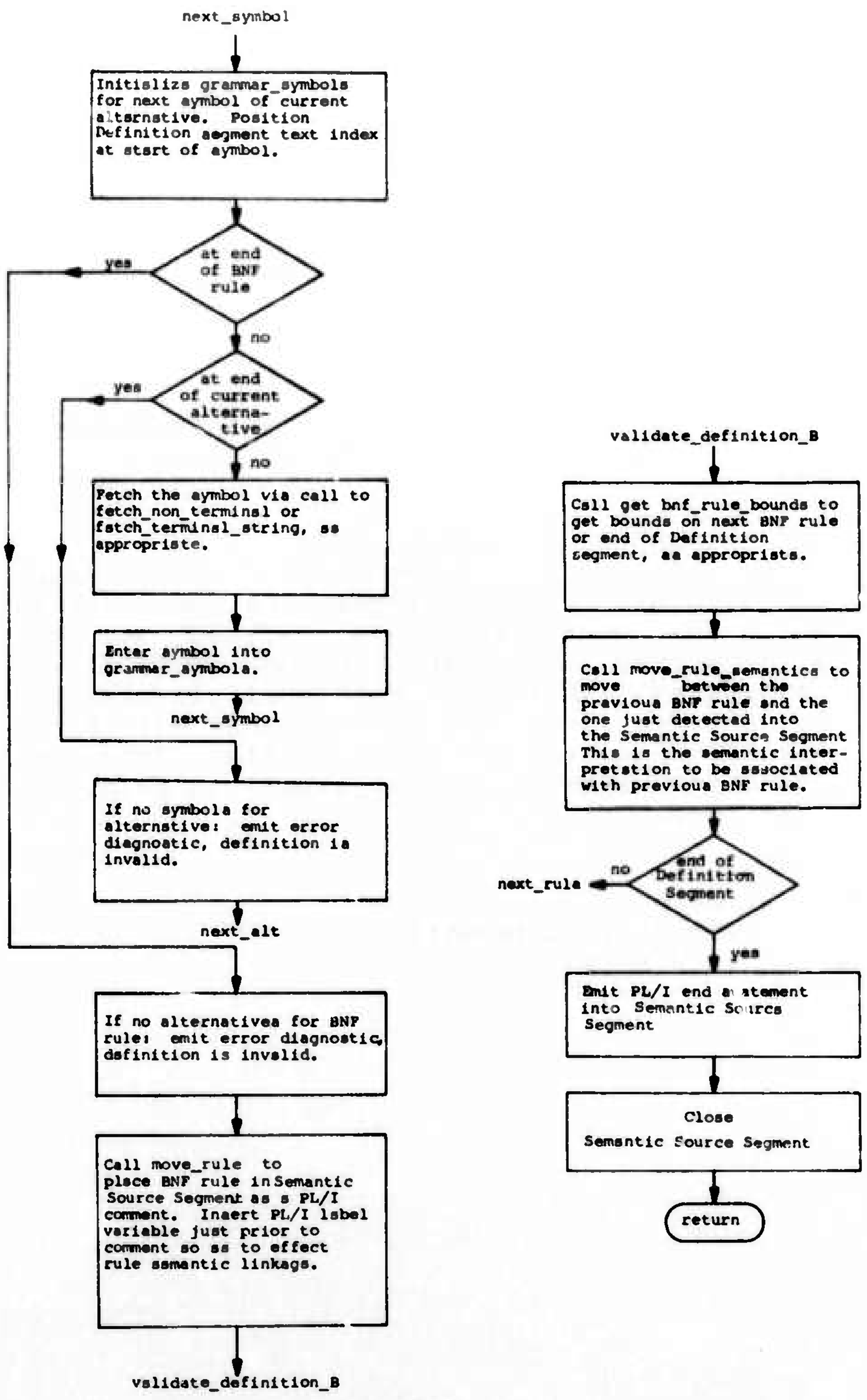
move_rule_semantics
This procedure moves the text of the LIS language Definition segment between the end of the current BNF rule and the start of the next BNF rule (or end of segment, if the current BNF rule is the last rule of the Definition) into the Semantic Source Segment as the semantics associated with the current rule. If the only characters of such text are blank, tab, new line, and new page characters, then there is assumed to be no meaningful semantics associated with the rule and the semantics bit of the rule is set to "0"b. Otherwise the semantics bit of the rule is set to "1"b. The text index is set to the start of the next BNF rule or end of segment, as appropriate.

```
Procedure Size:
   Source:  859 PL/I Statements
   Object:  7181 Words
```

validate_definition


entry
any arguments in call to lis
no
yes
Emit short description of argument formats, definition is invalid.
return
Check options, possibilities:
a) type of recognizer: primary, lexical, or both
b) request for LIS timing statistics
c) compute Semantic Source Seg. only
using lis_debug_monitor
no
yes
Check options, possibilities:
a) perform ambiguity check for simultaneous left and right recursion
b) print grammar structures
c) print configuration information
d) print CFSM state access list
e) print CFSM
f) print initial DPDA
Initialize LIS Language Definition
initialization successful
no
yes
Emit error diagnostic, definition is invalid.
return
Create and Open Semantic Source Segment
validate_definition_A
validate_definition_A
Create and open temporary files:
grammar_rules
grammar_alts
grammar_symbols
non_terminal_struct
initialization
Call get_bnf_rule_bounds to get bounds of first BNF rule
find first BNF rule
no
Emit error diagnostic, definition is invalid.
return
yes
Move text up to first BNF rule into Semantic Source Segment (Initialization Semantics)
Insert semantic linkage into Semantic Source Segment, immediately following initialization semantics
next_rule
Initialize grammar_rules for next BNF rule, and position segment text index at start of non-terminal on left side of BNF rule.
Call fetch_non_terminal to fetch the non_terminal left side of the BNF rule. Enter non_terminal into grammar_rules.
next_alt
Initialize grammar_alts for next alternative of BNF rule.
next_symbol

validate_definition (continued)


next_symbol
Initialize grammar_symbols for next symbol of current alternative. Position Definition segment text index at start of symbol.
at end of BNF rule
yes
no
at end of current alterna-tive
yes
no
Fetch the symbol via call to fetch_non_terminal or fetch_terminal_string, as appropriate.
Enter symbol into grammar_symbols.
next_symbol
If no symbols for alternative: emit error diagnostic, definition is invalid.
next_alt
If no alternatives for BNF rule: emit error diagnostic, definition is invalid.
Call move_rule to place BNF rule in Semantic Source Segment as a PL/I comment. Insert PL/I label variable just prior to comment so as to effect rule semantic linkage.
validate_definition_B
validate_definition_B
Call get bnf_rule_bounds to get bounds on next BNF rule or end of Definition segment, as appropriate.
Call move_rule_semantics to move between the previous BNF rule and the one just detected into the Semantic Source Segment This is the semantic inter-pretation to be associated with previous BNF rule.
end of Definition Segment
no
next_rule
yes
Emit PL/I end statement into Semantic Source Segment
Close Semantic Source Segment
return

B.2.3 <u>analyze_grammar</u>

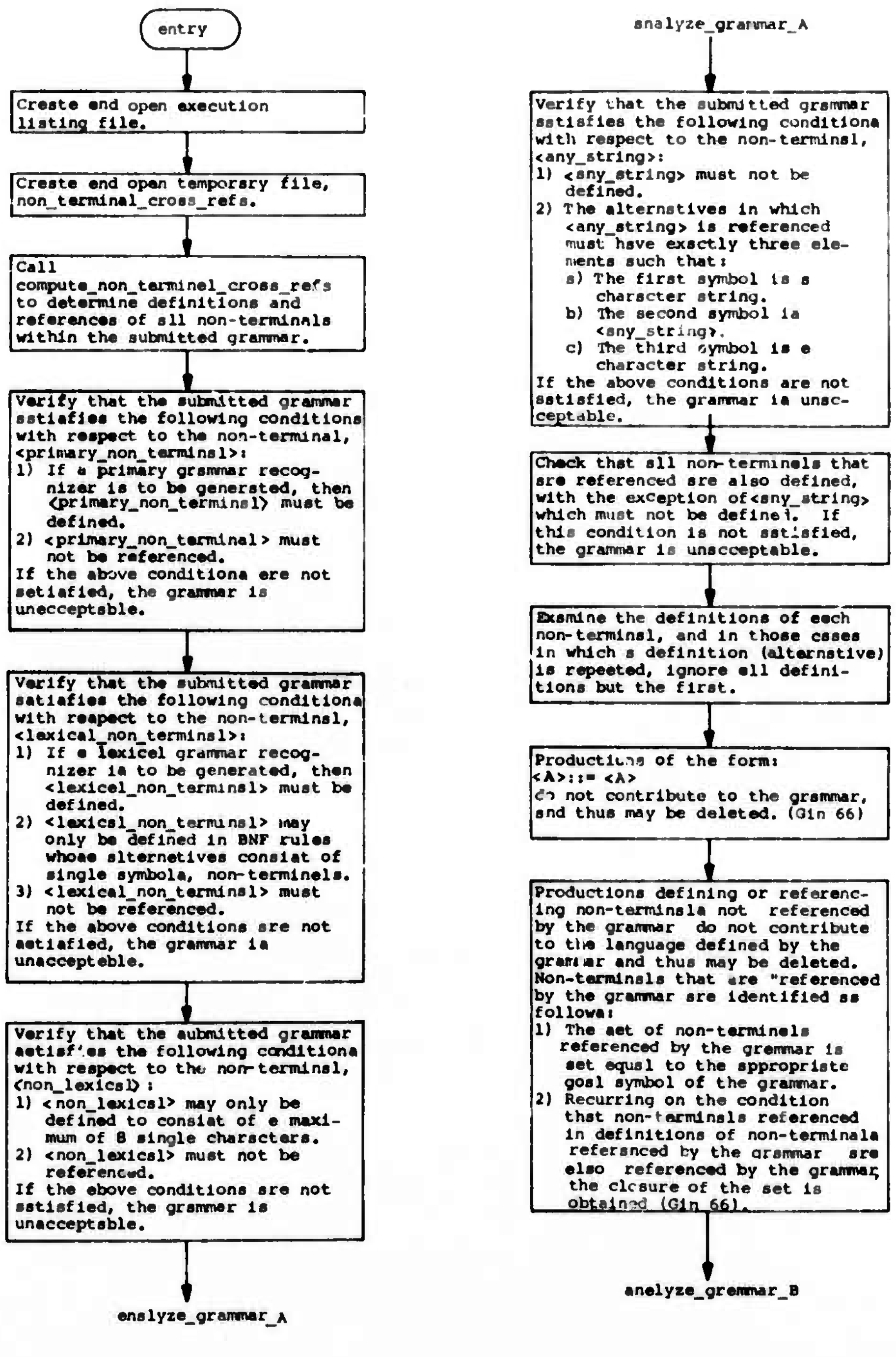
This procedure analyzes the submitted grammar to verify that certain conventions and linguistic structural requirements are satisfied. The implementation of the analysis procedures is indicated in the following flowcharts. A user oriented discussion of the conditions to be satisfied is given in Appendix A.

analyze_grammar makes use of the following internal procedure:

calculate_non_terminal_cross_refs
: This procedure determines, for each non-terminal in the grammar, the BNF rules in which the non-terminal is defined and the alternatives in which the non-terminal is referenced. This cross reference information is entered into the non-terminal structures (Section B.2) and is also written into the execution segment (".lis_exec") in printable text form.

Procedure Size:
Source: 954 PL/I Statements
Object: 7319 Words

analyze_grammar


entry

Create and open execution listing file.

Create and open temporary file, non_terminal_cross_refs.

Call compute_non_terminal_cross_refs to determine definitions and references of all non-terminals within the submitted grammar.

Verify that the submitted grammar satisfies the following conditions with respect to the non-terminal, <primary_non_terminal>:
1) If a primary grammar recognizer is to be generated, then <primary_non_terminal> must be defined.
2) <primary_non_terminal> must not be referenced.
If the above conditions are not satisfied, the grammar is unacceptable.

Verify that the submitted grammar satisfies the following conditions with respect to the non-terminal, <lexical_non_terminal>:
1) If a lexical grammar recognizer is to be generated, then <lexical_non_terminal> must be defined.
2) <lexical_non_terminal> may only be defined in BNF rules whose alternatives consist of single symbols, non-terminals.
3) <lexical_non_terminal> must not be referenced.
If the above conditions are not satisfied, the grammar is unacceptable.

Verify that the submitted grammar satisfies the following conditions with respect to the non-terminal, <non_lexical>:
1) <non_lexical> may only be defined to consist of a maximum of 8 single characters.
2) <non_lexical> must not be referenced.
If the above conditions are not satisfied, the grammar is unacceptable.

analyze_grammar_A

analyze_grammar_A

Verify that the submitted grammar satisfies the following conditions with respect to the non-terminal, <any_string>:
1) <any_string> must not be defined.
2) The alternatives in which <any_string> is referenced must have exactly three elements such that:
a) The first symbol is a character string.
b) The second symbol is <any_string>.
c) The third symbol is a character string.
If the above conditions are not satisfied, the grammar is unacceptable.

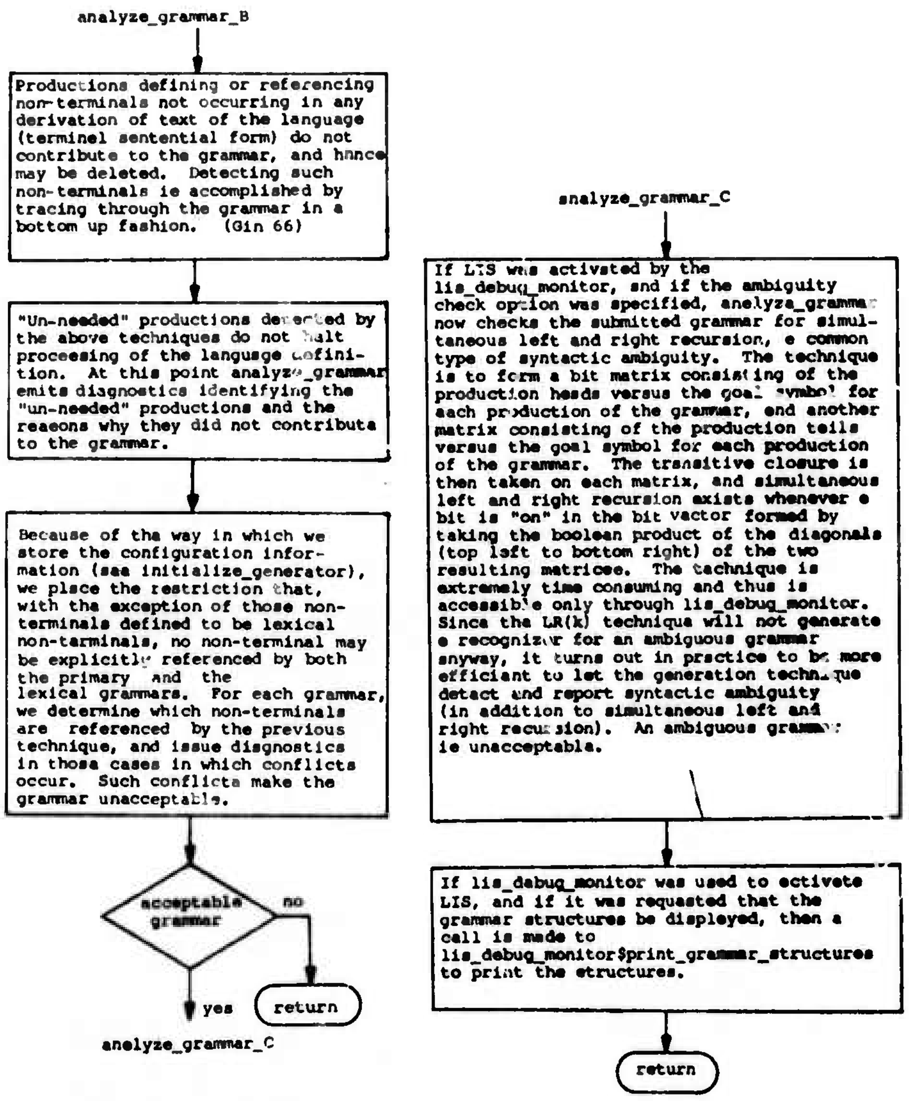
Check that all non-terminals that are referenced are also defined, with the exception of <any_string> which must not be defined. If this condition is not satisfied, the grammar is unacceptable.

Examine the definitions of each non-terminal, and in those cases in which a definition (alternative) is repeated, ignore all definitions but the first.

Productions of the form:
<A>::= <A>
do not contribute to the grammar, and thus may be deleted. (Gin 66)

Productions defining or referencing non-terminals not referenced by the grammar do not contribute to the language defined by the grammar and thus may be deleted. Non-terminals that are "referenced by the grammar are identified as follows:
1) The set of non-terminals referenced by the grammar is set equal to the appropriate goal symbol of the grammar.
2) Recurring on the condition that non-terminals referenced in definitions of non-terminals referenced by the grammar are also referenced by the grammar, the closure of the set is obtained (Gin 66).

analyze_grammar_B



analyze_grammar_B

Productions defining or referencing non-terminals not occurring in any derivation of text of the language (terminal sentential form) do not contribute to the grammar, and hence may be deleted. Detecting such non-terminals is accomplished by tracing through the grammar in a bottom up fashion. (Gin 66)

"Un-needed" productions detected by the above techniques do not halt processing of the language definition. At this point analyze_grammar emits diagnostics identifying the "un-needed" productions and the reasons why they did not contribute to the grammar.

Because of the way in which we store the configuration information (see initialize_generator), we place the restriction that, with the exception of those non-terminals defined to be lexical non-terminals, no non-terminal may be explicitly referenced by both the primary and the lexical grammars. For each grammar, we determine which non-terminals are referenced by the previous technique, and issue diagnostics in those cases in which conflicts occur. Such conflicts make the grammar unacceptable.

analyze_grammar_C

If LIS was activated by the lis_debug_monitor, and if the ambiguity check option was specified, analyze_grammar now checks the submitted grammar for simultaneous left and right recursion, a common type of syntactic ambiguity. The technique is to form a bit matrix consisting of the production heads versus the goal symbol for each production of the grammar, and another matrix consisting of the production tails versus the goal symbol for each production of the grammar. The transitive closure is then taken on each matrix, and simultaneous left and right recursion exists whenever a bit is "on" in the bit vector formed by taking the boolean product of the diagonals (top left to bottom right) of the two resulting matrices. The technique is extremely time consuming and thus is accessible only through lis_debug_monitor. Since the LR(k) technique will not generate a recognizer for an ambiguous grammar anyway, it turns out in practice to be more efficient to let the generation technique detect and report syntactic ambiguity (in addition to simultaneous left and right recursion). An ambiguous grammar is unacceptable.

If lis_debug_monitor was used to activate LIS, and if it was requested that the grammar structures be displayed, then a call is made to lis_debug_monitor$print_grammar_structures to print the structures.

B.2.4 initialize_generator

When activated at its primary entry point, initialize_generator performs the initialization necessary to compute the CLR(k) parser of the submitted grammar (primary or lexical). The initialization performed is as follows:

a) Various temporary files are created and opened (Section B.3).

b) The basic configuration information is computed and entered into the configuration information structures (Section B.3.4).

c) The apply configuration information is computed and entered into the configuration information structures.

The secondary entry points of initialize_generator are as follows:

enter_key_symbols
: This entry is used to create the key-symbol structures (Section B.3.3). When called with a key-symbol of the primary grammar, the entry scans the key-symbol structures to determine if the symbol has already been entered. If so, the index of the symbol within the structures is returned. If the key-symbol was not found in the structures, then it is appended to the structures, and the index that is returned is therefore the number of symbols in the structures.

fetch_effective_terminal_string
: This entry accepts a string of terminal characters from the LIS Language Definition segment (the characters constituting either a non-terminal or a terminal symbol of the grammar) and converts it to a canonical form by resolving all escaped characters in the

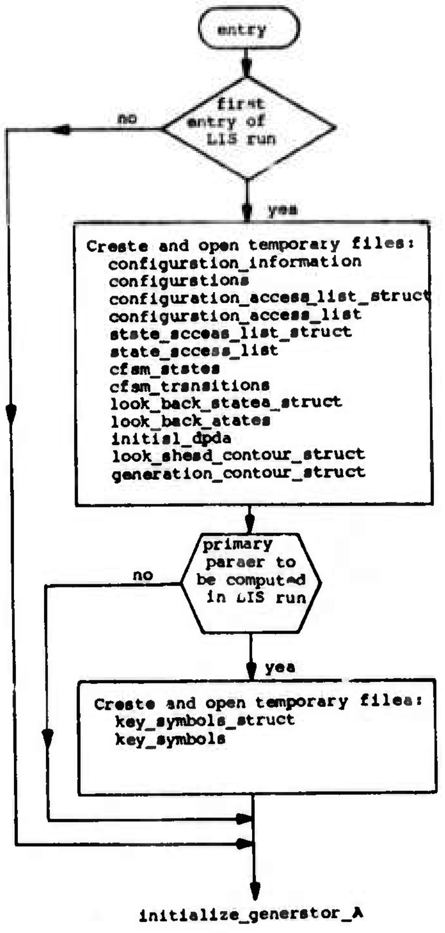
string.

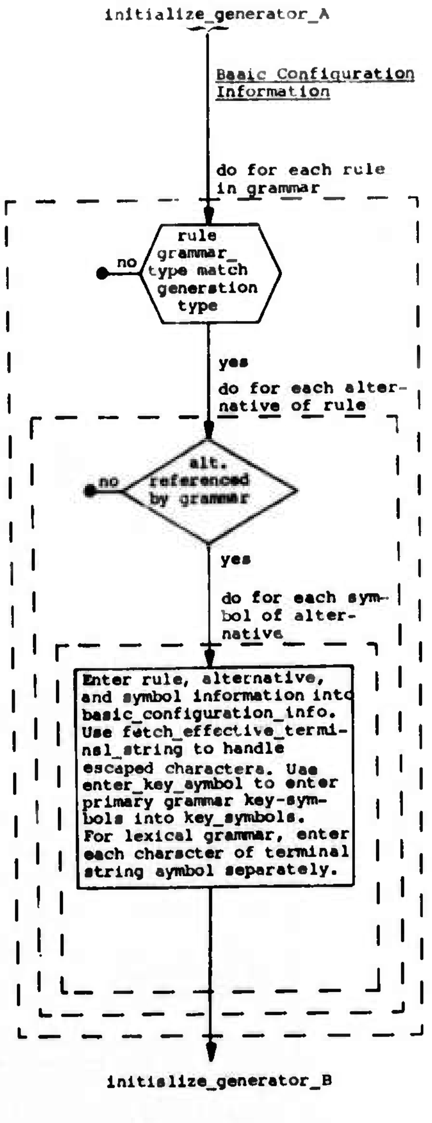
Procedure Size:

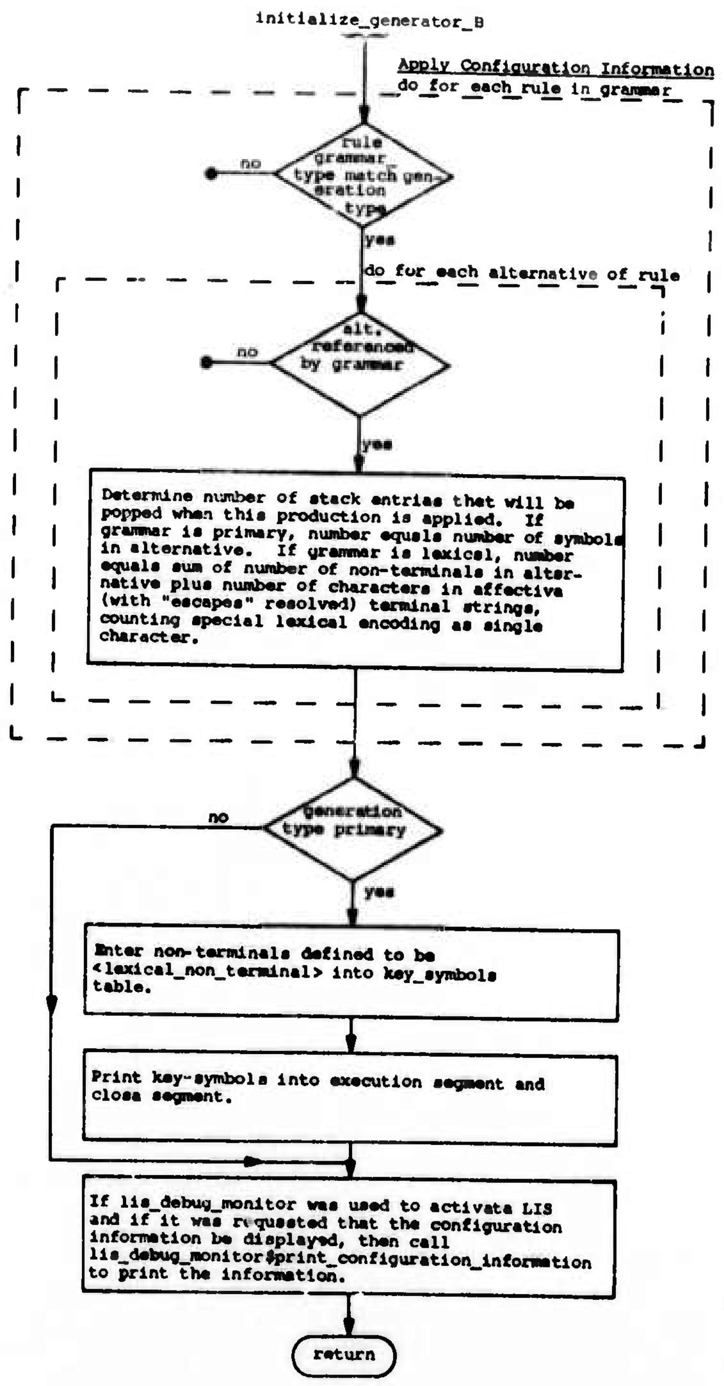
Source: 502 PL/I Statements

Object: 4447 Words

initialize_generator


entry
first entry of LIS run
no
yes
Create and open temporary files:
configuration_information
configurations
configuration_access_list_struct
configuration_access_list
state_access_list_struct
state_access_list
cfsm_states
cfsm_transitions
look_back_states_struct
look_back_states
initial_dpda
look_ahead_contour_struct
generation_contour_struct
primary parser to be computed in LIS run
no
yes
Create and open temporary files:
key_symbols_struct
key_symbols
initialize_generator_A


initialize_generator_A
Basic Configuration Information
do for each rule in grammar
rule grammar_type match generation type
no
yes
do for each alternative of rule
alt. referenced by grammar
no
yes
do for each symbol of alternative
Enter rule, alternative, and symbol information into basic_configuration_info. Use fetch_effective_terminal_string to handle escaped characters. Use enter_key_symbol to enter primary grammar key-symbols into key_symbols. For lexical grammar, enter each character of terminal string symbol separately.
initialize_generator_B


initialize_generator_B
Apply Configuration Information
do for each rule in grammar
rule grammar type match generation type
no
yes
do for each alternative of rule
alt. referenced by grammar
no
yes
Determine number of stack entries that will be popped when this production is applied. If grammar is primary, number equals number of symbols in alternative. If grammar is lexical, number equals sum of number of non-terminals in alternative plus number of characters in affective (with "escapes" resolved) terminal strings, counting special lexical encoding as single character.
generation type primary
no
yes
Enter non-terminals defined to be <lexical_non_terminal> into key_symbols table.
Print key-symbols into execution segment and close segment.
If lis_debug_monitor was used to activate LIS and if it was requested that the configuration information be displayed, then call lis_debug_monitor$print_configuration_information to print the information.
return

B.2.5 compute_cfsm

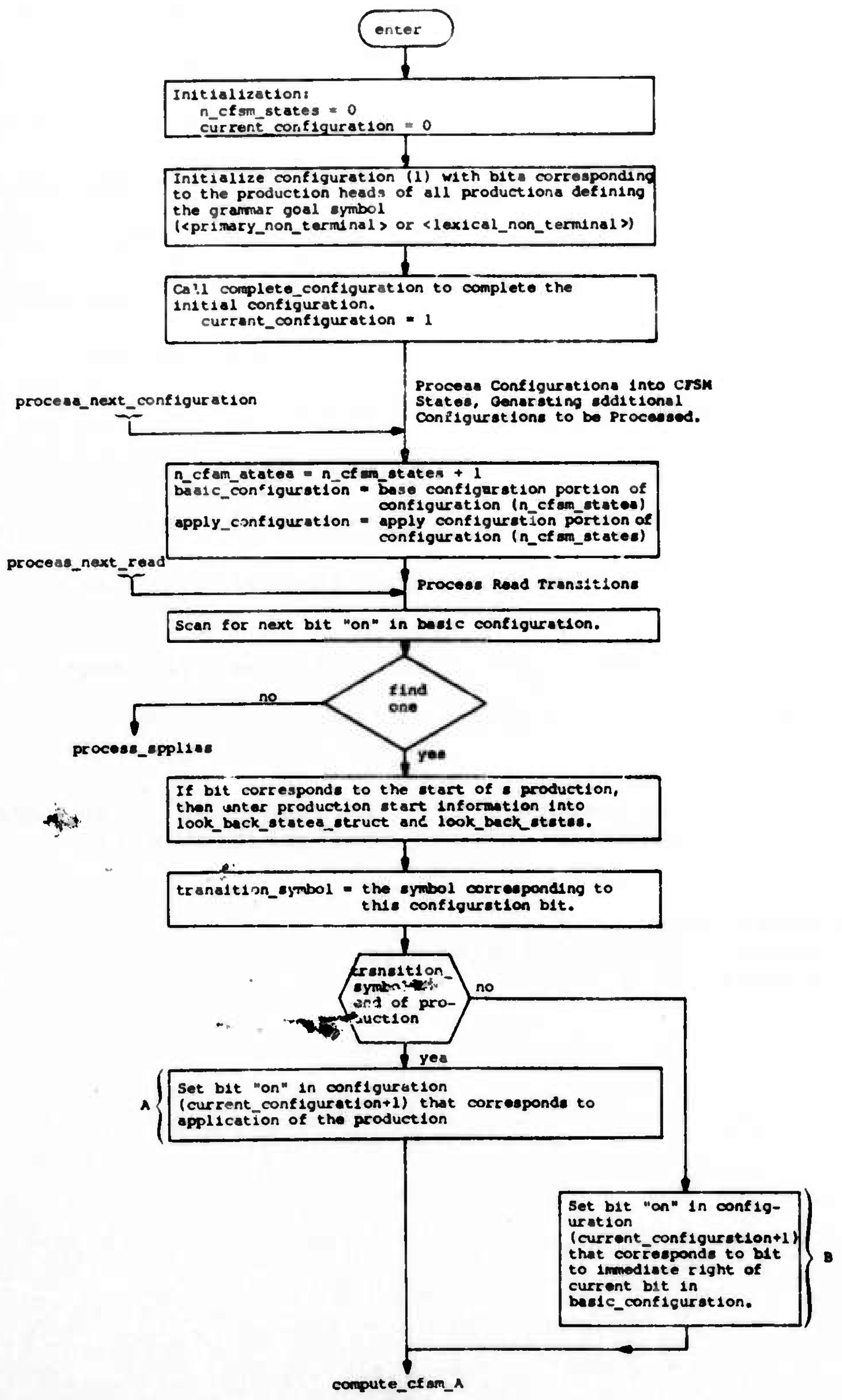
This procedure accepts the grammar structures (Section B.2.1), the non-terminal structures (Section B.3.2), the key-symbol structures (Section B.3.3), and the configuration information structures (Section B.3.4), and computes the Characteristic Finite State Machine (CFSM) from the specified grammar (primary or lexical). The algorithm is that of Knuth-Early (Ear 70), modified so as to compute either lexical or primary parsers by the method of iterative computation of configurations The representation and interpretation of the configurations is discussed in Sections B.3.4 and B.3.5. (See Chapter III, Section III.C).

compute_cfsm makes use of the following internal procedure:

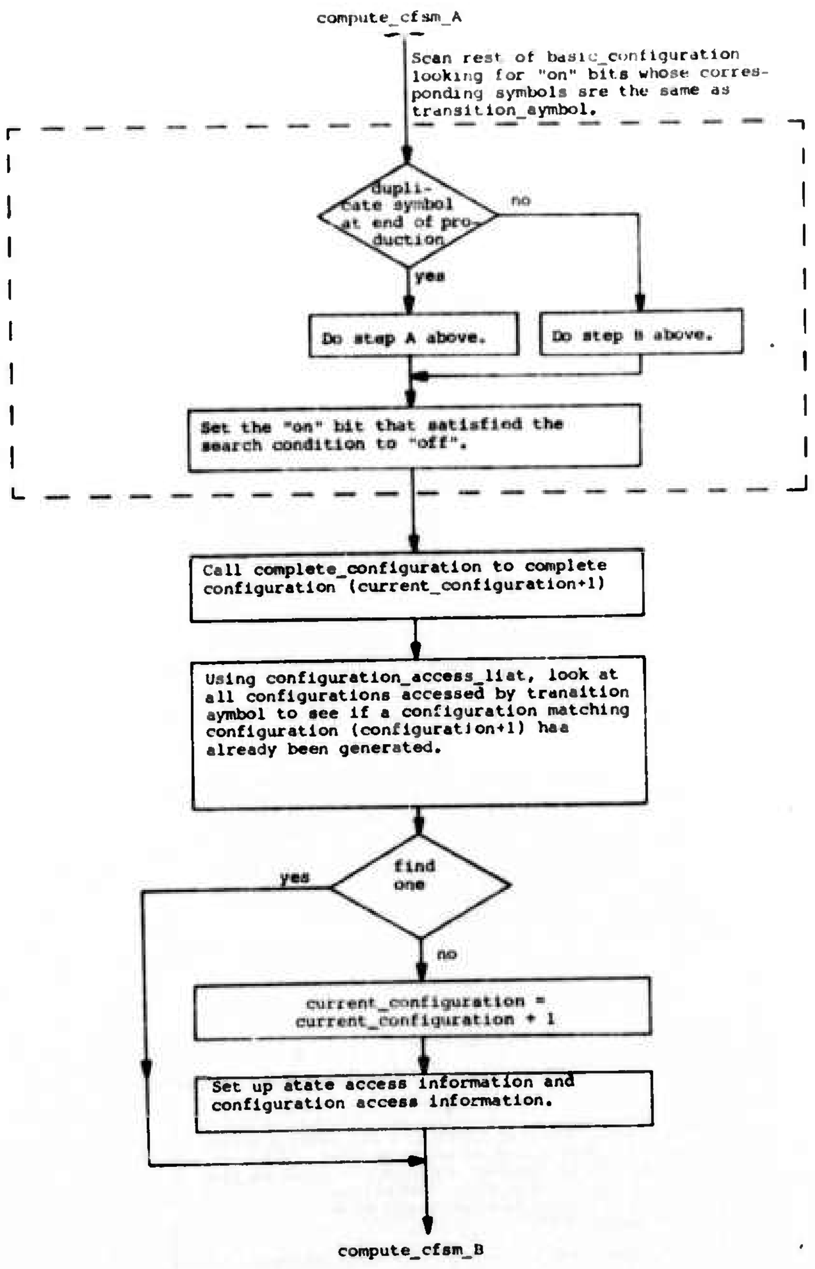
```
    complete_configuration
       This procedure is invoked to complete
       configuration(current_configuration + 1).
```

```
Procedure Size:
   Source:  512 PL/I Statements
   Object:  2873 Words
```

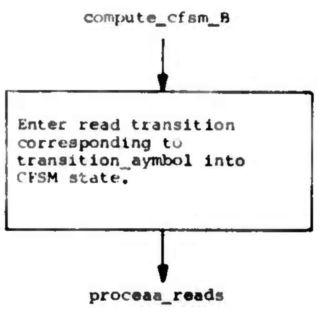
compute_cfsm



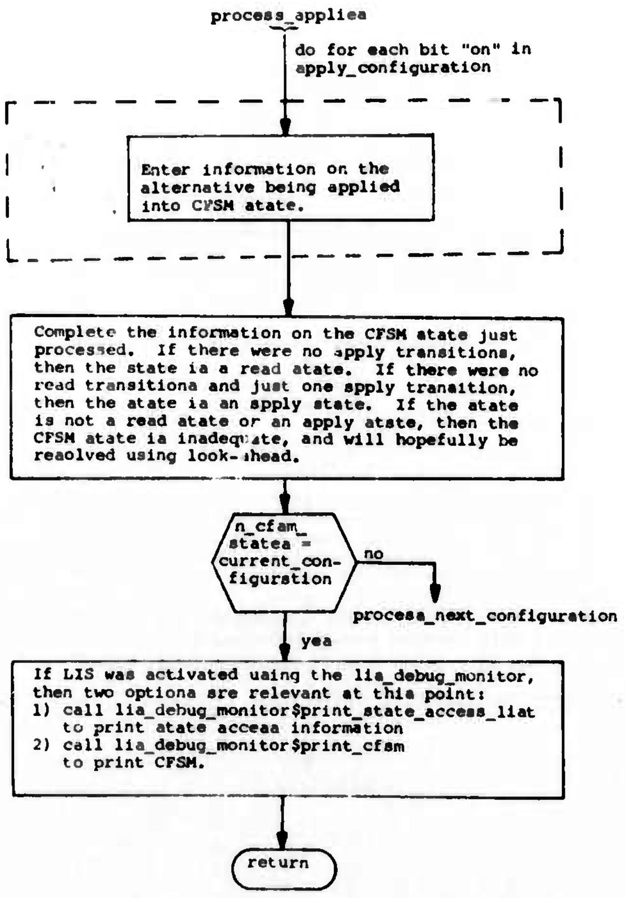
enter
Initialization:
n_cfsm_states = 0
current_configuration = 0
Initialize configuration (1) with bits corresponding to the production heads of all productions defining the grammar goal symbol (<primary_non_terminal> or <lexical_non_terminal>)
Call complete_configuration to complete the initial configuration.
currant_configuration = 1
Process Configurations into CFSM States, Generating additional Configurations to be Processed.
process_next_configuration
n_cfsm_states = n_cfsm_states + 1
basic_configuration = base configuration portion of configuration (n_cfsm_states)
apply_configuration = apply configuration portion of configuration (n_cfsm_states)
process_next_read
Process Read Transitions
Scan for next bit "on" in basic configuration.
find one
no
yes
process_applies
If bit corresponds to the start of a production, then enter production start information into look_back_states_struct and look_back_states.
transition_symbol = the symbol corresponding to this configuration bit.
transition_symbol = end of production
no
yes
Set bit "on" in configuration (current_configuration+1) that corresponds to application of the production
A
Set bit "on" in configuration (current_configuration+1) that corresponds to bit to immediate right of current bit in basic_configuration.
B
compute_cfsm_A


compute_cfsm_A
Scan rest of basic_configuration looking for "on" bits whose corresponding symbols are the same as transition_symbol.
duplicate symbol at end of production
no
yes
Do step A above.
Do step B above.
Set the "on" bit that satisfied the search condition to "off".
Call complete_configuration to complete configuration (current_configuration+1)
Using configuration_access_list, look at all configurations accessed by transition symbol to see if a configuration matching configuration (configuration+1) has already been generated.
find one
yes
no
current_configuration = current_configuration + 1
Set up state access information and configuration access information.
compute_cfsm_B

compute_cfsm (continued)



Enter read transition corresponding to transition_symbol into CFSM state.


process_applies
do for each bit "on" in apply_configuration
n_cfsm_states = current_configuration
no
process_next_configuration
yes
return

Enter information on the alternative being applied into CFSM state.

Complete the information on the CFSM state just processed. If there were no apply transitions, then the state is a read state. If there were no read transitions and just one apply transition, then the state is an apply state. If the state is not a read state or an apply state, then the CFSM state is inadequate, and will hopefully be resolved using look-ahead.

If LIS was activated using the lis_debug_monitor, then two options are relevant at this point:
1) call lis_debug_monitor$print_state_access_list to print state access information
2) call lis_debug_monitor$print_cfsm to print CFSM.

B.2.6 convert_cfsm_to_dpda

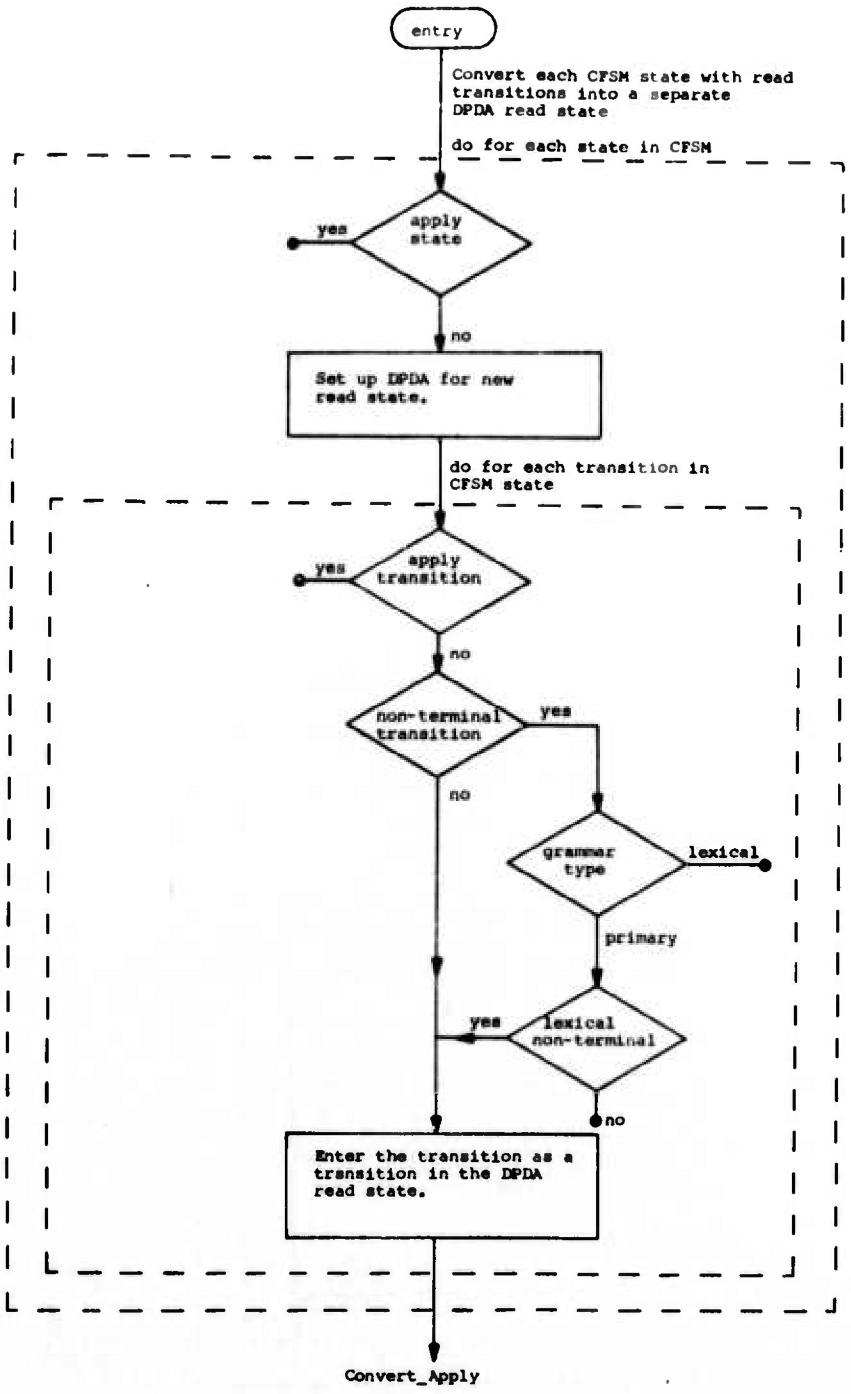
This procedure converts the CFSM computed by compute_cfsm into a Deterministic Push Down Automaton (DPDA) with look-ahead, and enters the DPDA into initial_dpda (Section B.3.10). First the procedure converts each CFSM state with read transitions into a separate DPDA read state. In so doing, all read transitions on non-terminals are deleted, except that when converting a primary CFSM, those non-terminals that are <lexical_non_terminal>s are retained.

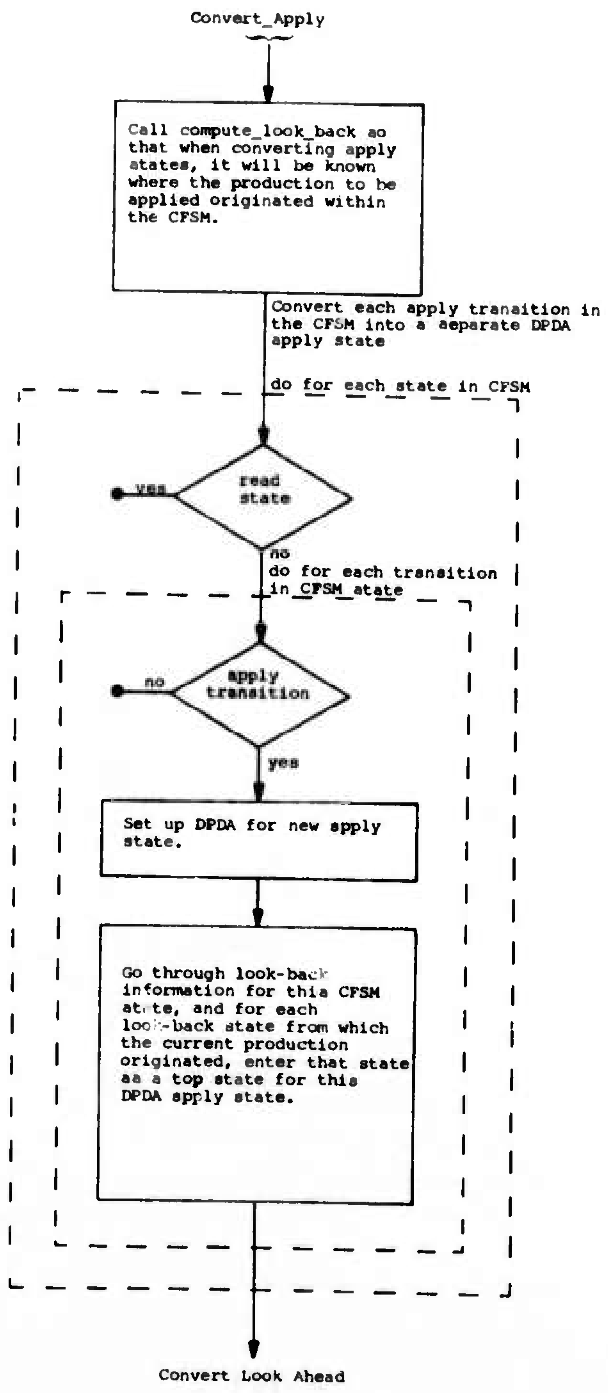
As a first step in the conversion of the apply states, the internal procedure, compute_look_back, is invoked to determine, for each CFSM state containing at least one apply transition, the set of look-back states that is appropriate to the production(s) being applied in the state (see discussion of look-back structures, Section B.3.9). Following the computation of the look-back sets, each apply transition of the CFSM is converted into a separate DPDA apply state. In converting each transition, a search is made of the look-back set associated with the CFSM state to which the transition belongs, and an entry is made in the corresponding DPDA apply state's list of top states for each such look-back state of the set from which the production being applied may have originated.

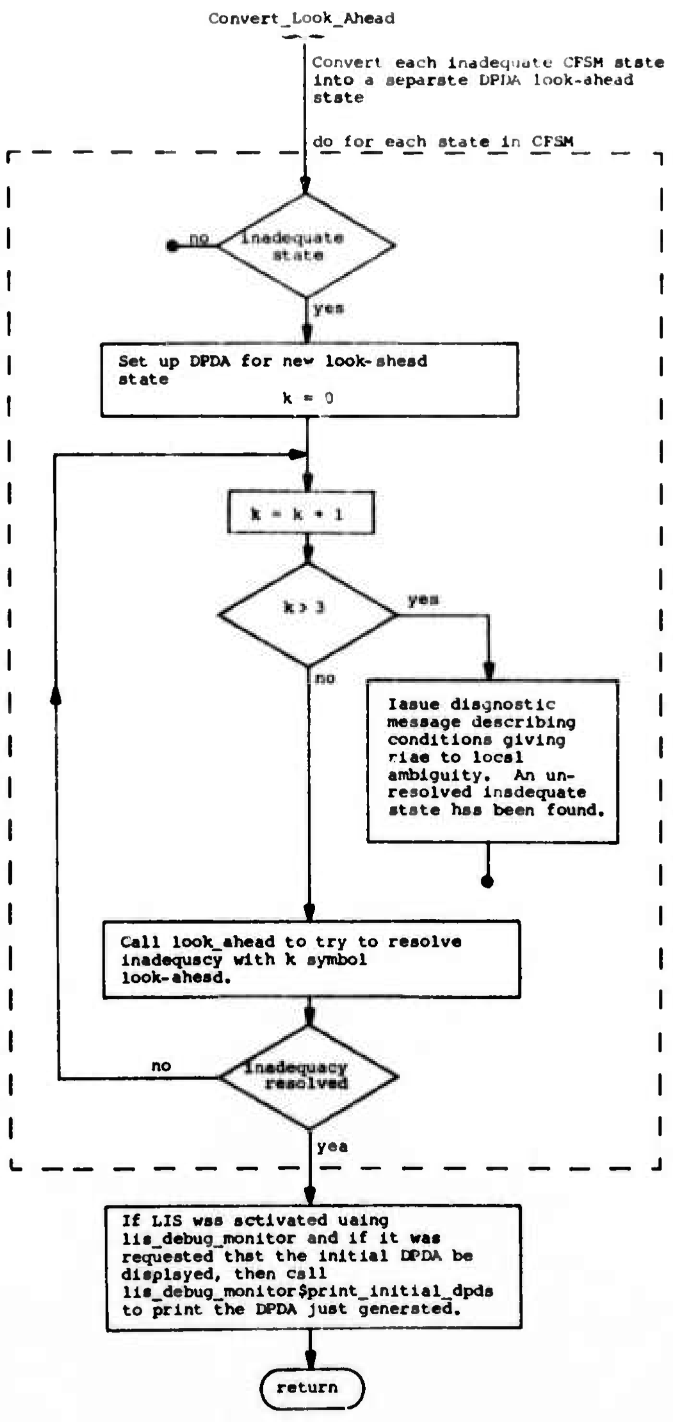
For each inadequate state of the CFSM, convert_cfsm_to_dpda attempts to resolve the inadequacy by converting the state into a DPDA look-ahead state. Resolution is attempted by looking ahead a maximum of three symbols. The external procedure, look_ahead (see Section B.2.7), is invoked for purposes of determining look-ahead symbols and the internal procedure, intersecting_look_ahead_sets, is invoked to determine if the inadequacy has been resolved.

```
Procedure Size:
   Source:  602 PL/I Statements
   Object:  3325 Words
```

convert_cfsm_to_dpda


entry
Convert each CFSM state with read transitions into a separate DPDA read state
do for each state in CFSM
apply state
yes
no
Set up DPDA for new read state.
do for each transition in CFSM state
apply transition
yes
no
non-terminal transition
yes
no
grammar type
lexical
primary
lexical non-terminal
yes
no
Enter the transition as a transition in the DPDA read state.
Convert_Apply


Convert_Apply
Call compute_look_back so that when converting apply states, it will be known where the production to be applied originated within the CFSM.
Convert each apply transition in the CFSM into a separate DPDA apply state
do for each state in CFSM
read state
yes
no
do for each transition in CFSM state
apply transition
no
yes
Set up DPDA for new apply state.
Go through look-back information for this CFSM state, and for each look-back state from which the current production originated, enter that state as a top state for this DPDA apply state.
Convert Look Ahead


Convert_Look_Ahead
Convert each inadequate CFSM state into a separate DPDA look-ahead state
do for each state in CFSM
inadequate state
no
yes
Set up DPDA for new look-ahead state
k = 0
k = k + 1
k > 3
yes
no
Issue diagnostic message describing conditions giving rise to local ambiguity. An unresolved inadequate state has been found.
Call look_ahead to try to resolve inadequacy with k symbol look-ahead.
inadequacy resolved
no
yes
If LIS was activated using lis_debug_monitor and if it was requested that the initial DPDA be displayed, then call lis_debug_monitor$print_initial_dpda to print the DPDA just generated.
return

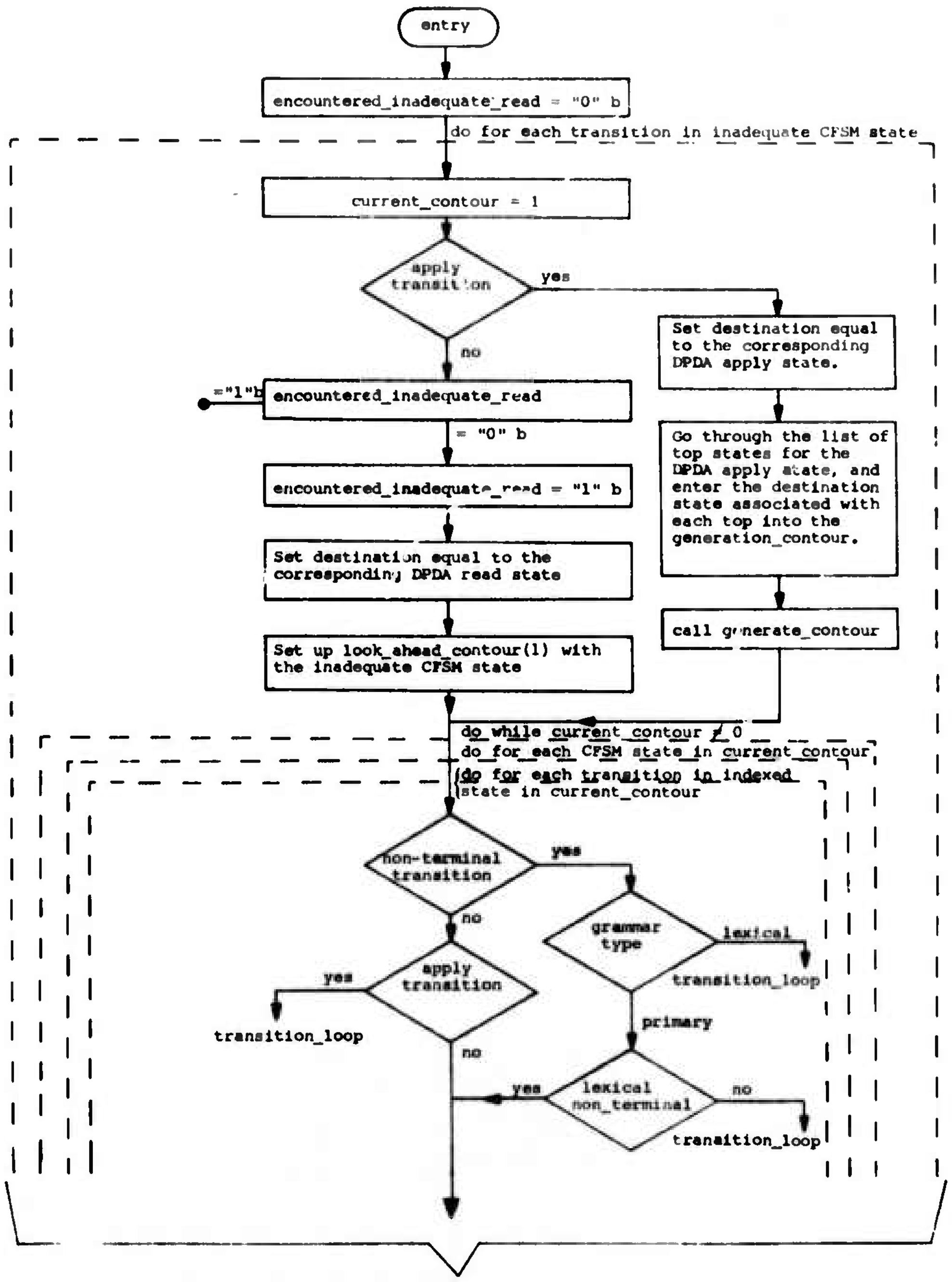
B.2.7 look_ahead

This procedure is invoked to compute k symbols of look-ahead ($k \leq 3$) for the specified inadequate CFSM state. In performing the look-ahead, the procedure makes use of the following internal procedure:

generate_contour
: This procedure expands an initial generation_contour and enters the resulting read and look-ahead states into look_ahead_contour(current_contour) (see discussion of look-ahead structures, Section B.2.11).

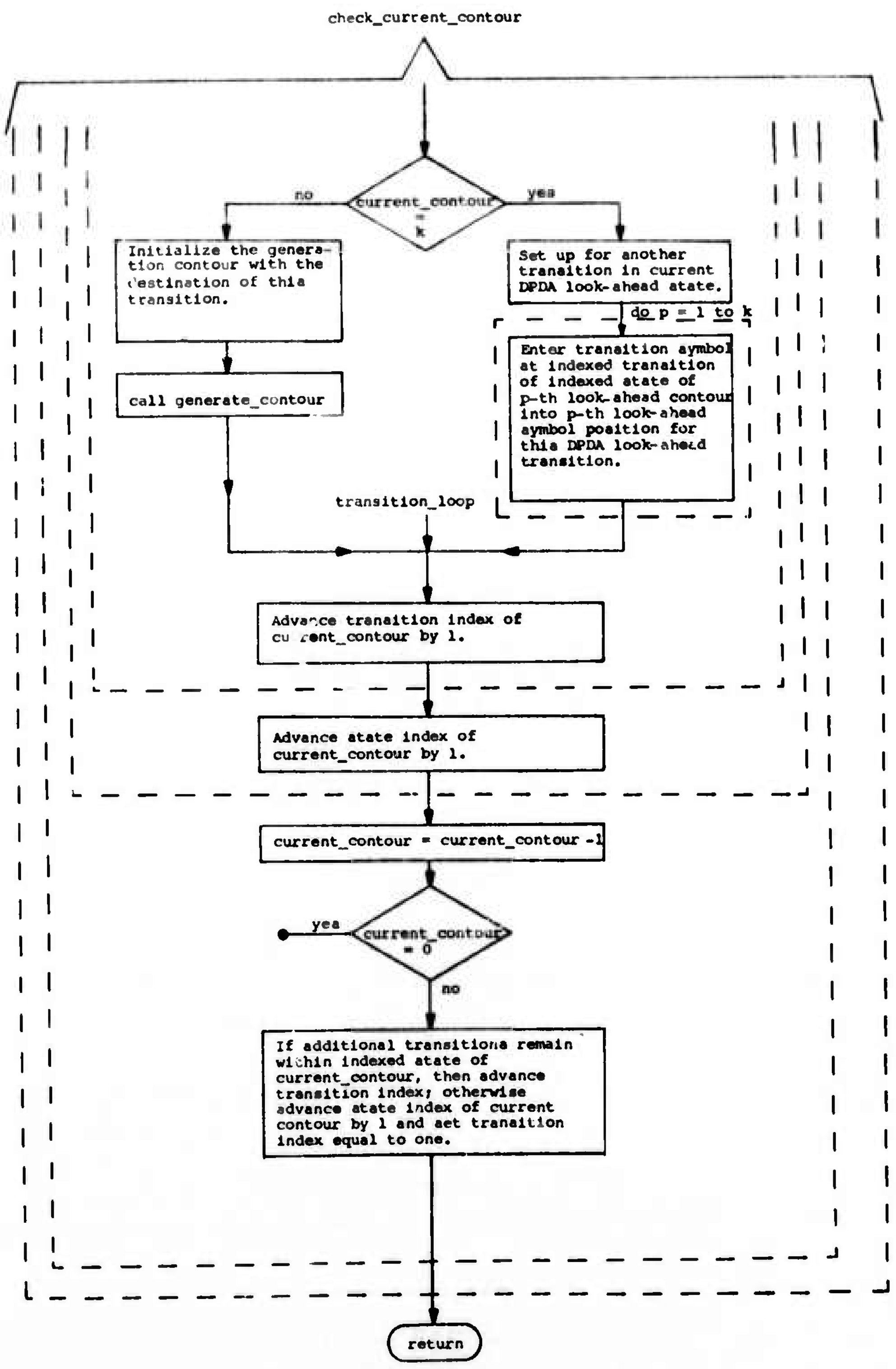
Procedure Size:
- Source: 335 PL/I Statements
- Object: 2084 Words

look_ahead


entry
encountered_inadequate_read = "0" b
do for each transition in inadequate CFSM state
current_contour = 1
apply transition
yes
Set destination equal to the corresponding DPDA apply state.
Go through the list of top states for the DPDA apply state, and enter the destination state associated with each top into the generation_contour.
call generate_contour
no
="1"b
encountered_inadequate_read
= "0" b
encountered_inadequate_read = "1" b
Set destination equal to the corresponding DPDA read state
Set up look_ahead_contour(1) with the inadequate CFSM state
do while current_contour ≠ 0
do for each CFSM state in current_contour
do for each transition in indexed state in current_contour
non-terminal transition
yes
grammar type
lexical
transition_loop
no
apply transition
yes
transition_loop
no
primary
lexical non_terminal
yes
no
transition_loop

check_current_contour

look_ahead (continued)


check_current_contour
current_contour = k
no
yes
Initialize the generation contour with the destination of this transition.
call generate_contour
Set up for another transition in current DPDA look-ahead state.
do p = 1 to k
Enter transition symbol at indexed transition of indexed state of p-th look-ahead contour into p-th look-ahead symbol position for this DPDA look-ahead transition.
transition_loop
Advance transition index of current_contour by 1.
Advance state index of current_contour by 1.
current_contour = current_contour -1
current_contour = 0
yes
no
If additional transitions remain within indexed state of current_contour, then advance transition index; otherwise advance state index of current contour by 1 and set transition index equal to one.
return

B.2.8 <u>optimize_dpda</u>

This procedure optimizes the <u>contents</u> of the DPDA produced by convert_cfsm_to_dpda, and enters the optimized DPDA into final_dpda (Section B.3.12). Transformations are performed on the DPDA that remove superfluous and redundant information so that the resulting DPDA is more efficient than its predecessor. The optimizations that have been implemented by no means exhaust the potential for DPDA content optimization. Other optimizations, such as transition sorting according to empirical measures of frequency of transition occurrence for a particular language, transition hashing, detection and deletion of apply states that have no associated semantics and that do not modify the DPDA state stack, are but a few of the optimizations that have not been implemented on LIS, but that could have significant impact on parser space and time efficiency.

The representation of the DPDA read states is such that the information regarding the states themselves is stored separately from the information on the state's transitions. This being the case, we can optimize the read transitions by deleting duplicate transition sequences that may arise from different read states.

There are two fundamental optimizations that are performed on the DPDA apply states. First, for each apply state, we determine the most popular look-back transition destination state, and designate that the default destination state. Then all look-back transitions (also called to transitions or apply transitions) of the state whose destination state is the default destination state are deleted from the list of look-back transitions. The default destination state is then appended to the list, it being the convention that, during the language recognition process, should the top of the state stack (after being popped) fail to match any of the look-back states in the list for the current apply state, then the transition to the default destination state is automatically taken. Since the LR(k) recognition process is deterministic, we are guaranteed not to introduce any errors into the recognition process by performing this optimization.

The second optimization that we apply to the DPDA apply states is analogous to the optimization applied to the DPDA read states, and we thus remove redundant information.

The number of optimizations performed on the DPDA look-ahead states is one or two, depending on whether the grammar in question is lexical or primary, respectively. In either case, duplicate look-ahead transitions for a given look-ahead state are deleted. In the case of a parser computed from a primary grammar, an additional optimization is performed on the look-ahead states which is analogous to the first of the optimizations applied to the DPDA apply states. Thus, for each look-ahead state, we determine the most popular look-ahead transition destination state, and designate that the default destination state. Any look-ahead transition whose destination state matches the default destination state is deleted from the list of look-ahead transitions for the state in question, and the default destination state is appended to the end of the list. The recognition time interpretation of the default look-ahead transition is analogous to the recognition time interpretation of the default apply transition. However, in the present case, the detection of an erroneous symbol in the input stream will be delayed until a subsequent read state, whereas were the default destination optimization not performed, such an error would be detected in the look-ahead state.

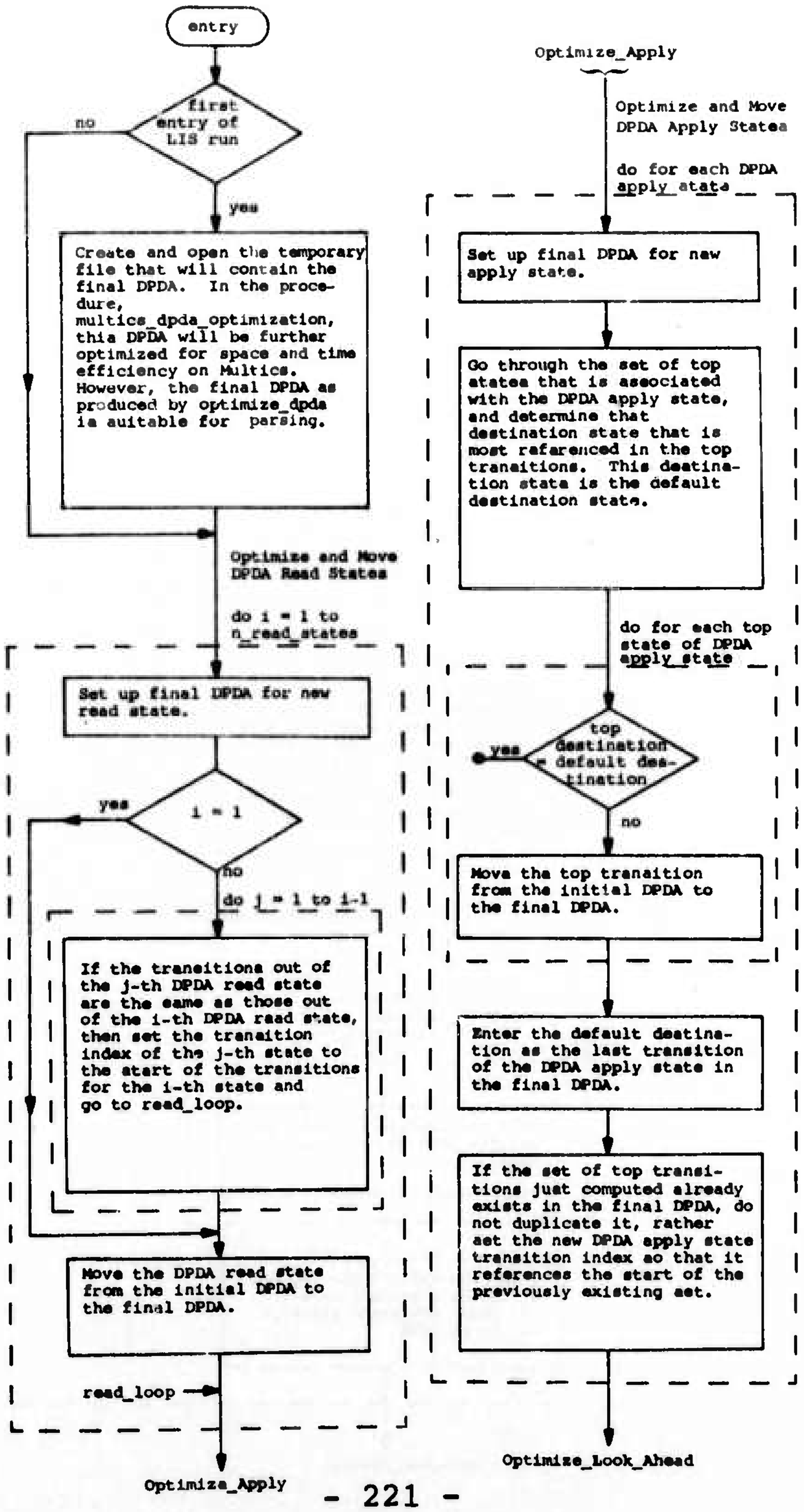
In addition to performing the above optimizations, optimize_dpda moves the key-symbol structures into the optimized DPDA for those DPDAs computed from a primary grammar.

optimize_dpda makes use of the following internal procedure:

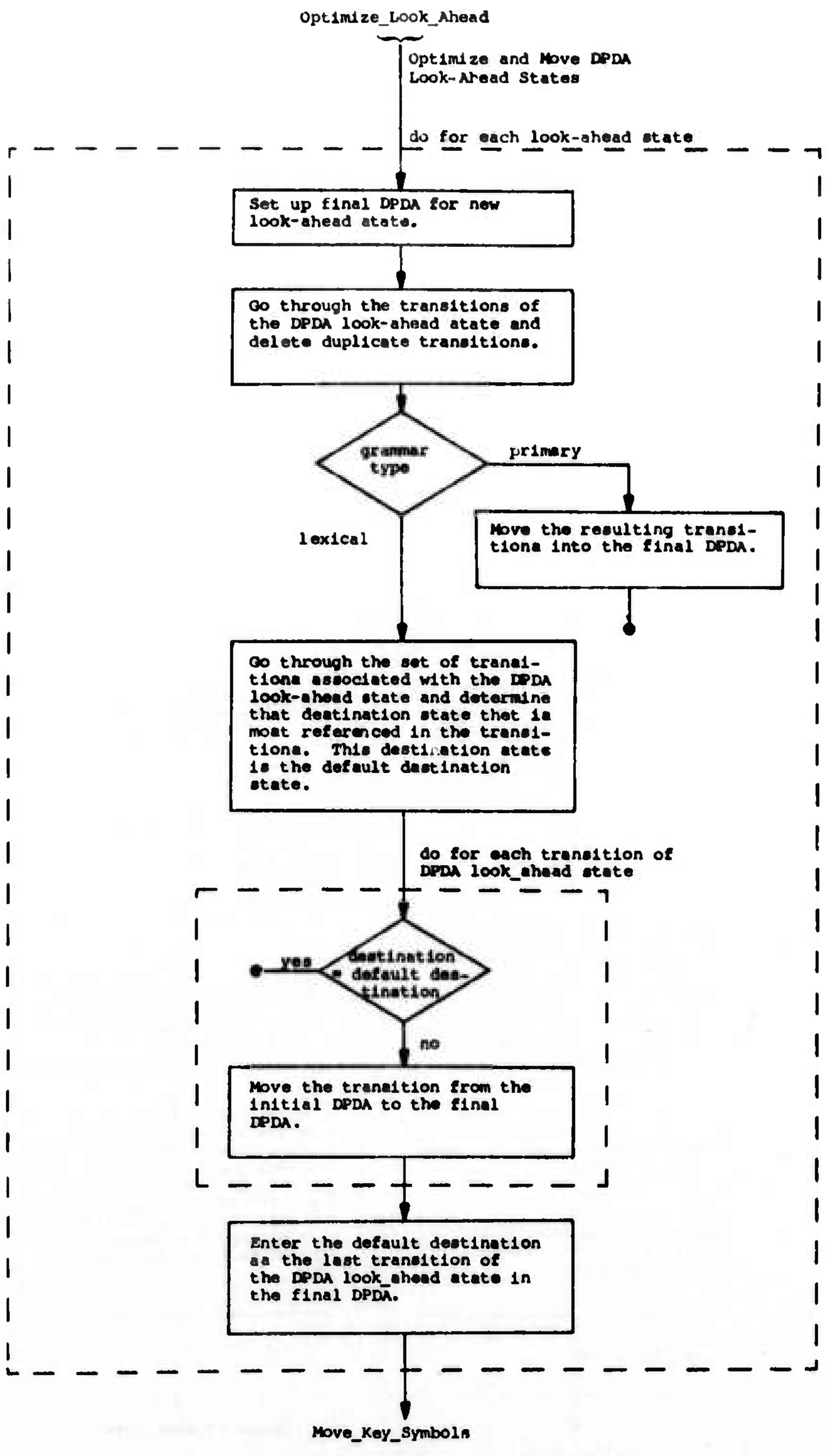
print_dpda
This procedure is invoked after the DPDA has been optimized (and the key-svmbol structures moved, if appropriate) and simply writes the optimized DPDA into the ".lis_dpda" segment in printable text form.

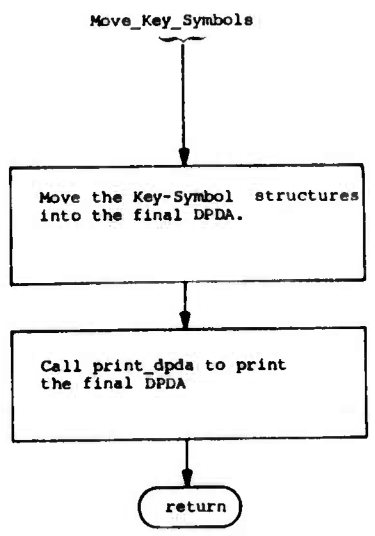
Procedure Size:
Source: 950 PL/I Statements
Object: 7584 Words

optimize_dpda


entry
first entry of LIS run
no
yes
Create and open the temporary file that will contain the final DPDA. In the procedure, multics_dpda_optimization, this DPDA will be further optimized for space and time efficiency on Multics. However, the final DPDA as produced by optimize_dpda is suitable for parsing.
Optimize and Move DPDA Read States
do i = 1 to n_read_states
Set up final DPDA for new read state.
i = 1
yes
no
do j = 1 to i-1
If the transitions out of the j-th DPDA read state are the same as those out of the i-th DPDA read state, then set the transition index of the j-th state to the start of the transitions for the i-th state and go to read_loop.
Move the DPDA read state from the initial DPDA to the final DPDA.
read_loop
Optimize_Apply
Optimize_Apply
Optimize and Move DPDA Apply States
do for each DPDA apply state
Set up final DPDA for new apply state.
Go through the set of top states that is associated with the DPDA apply state, and determine that destination state that is most referenced in the top transitions. This destination state is the default destination state.
do for each top state of DPDA apply state
top destination = default destination
yes
no
Move the top transition from the initial DPDA to the final DPDA.
Enter the default destination as the last transition of the DPDA apply state in the final DPDA.
If the set of top transitions just computed already exists in the final DPDA, do not duplicate it, rather set the new DPDA apply state transition index so that it references the start of the previously existing set.
Optimize_Look_Ahead

optimize_dpda (continued)


Optimize_Look_Ahead
Optimize and Move DPDA Look-Ahead States
do for each look-ahead state
Set up final DPDA for new look-ahead state.
Go through the transitions of the DPDA look-ahead state and delete duplicate transitions.
grammar type
primary
lexical
Move the resulting transitions into the final DPDA.
Go through the set of transitions associated with the DPDA look-ahead state and determine that destination state that is most referenced in the transitions. This destination state is the default destination state.
do for each transition of DPDA look_ahead state
yes
destination = default destination
no
Move the transition from the initial DPDA to the final DPDA.
Enter the default destination as the last transition of the DPDA look_ahead state in the final DPDA.
Move_Key_Symbols


Move_Key_Symbols
Move the Key-Symbol structures into the final DPDA.
Call print_dpda to print the final DPDA
return

B.2.9 <u>multics_dpda_optimization</u>

This procedure optimizes the <u>representation</u> of the final DPDA on the Multics system, and in so doing, enhances the space and time efficiency of the parsers that execute on Multics. The optimizations performed are representative of the type of "fine tuning" that should be considered when LIS is used to produce parsers for a particular computing environment. Being only representative, they by no means exhaust the type of optimizations that can be performed, even on Multics. The possibilities for machine dependent DPDA optimization are limited only by one's imagination and understanding of the space-time tradeoffs inherent in the computing environment under consideration.

The optimizations performed by this procedure are straightforward, and will not be considered in further detail. A comparison of the Final DPDA Structures (Appendix B.3.12) with the Multics DPDA Structures (Appendix B.3.13) should give the reader an understanding of the DPDA optimizations that have been implemented.

Procedure Size:
   Source: 650 PL/I Statements
   Object: 5213 Words

B.2.10 lis_debug_monitor

This procedure may be invoked instead of the lis procedure in those cases in which the LIS system is being modified and debugged. The procedure performs some simple initializations related to its debugging options, and immediately invokes the lis procedure. Thereafter, the LIS system responds to the specified debugging requests by invoking the appropriate secondary entry points below.

print_grammar_structures
: This entry prints the grammar structures (Section B.3.1) into the ".lis_debug" file.

print_configuration_information
: This entry prints the configuration information structures (Section B.3.4) into the ".lis_debug" file.

print_cfsm
: This entry prints the CFSM structures (Section B.3.7) into the ".lis_cfsm" file.

print_state_access
: This entry prints the state access structures (Section B.3.8) into the ".lis_debug" file.

print_initial_dpda
: This entry prints the initial DPDA strucures (Section B.3.10) into the ".lis_init_dpda" file.

Procedure Size:
  Source: 622 PL/I Statements
  Object: 5834 Words

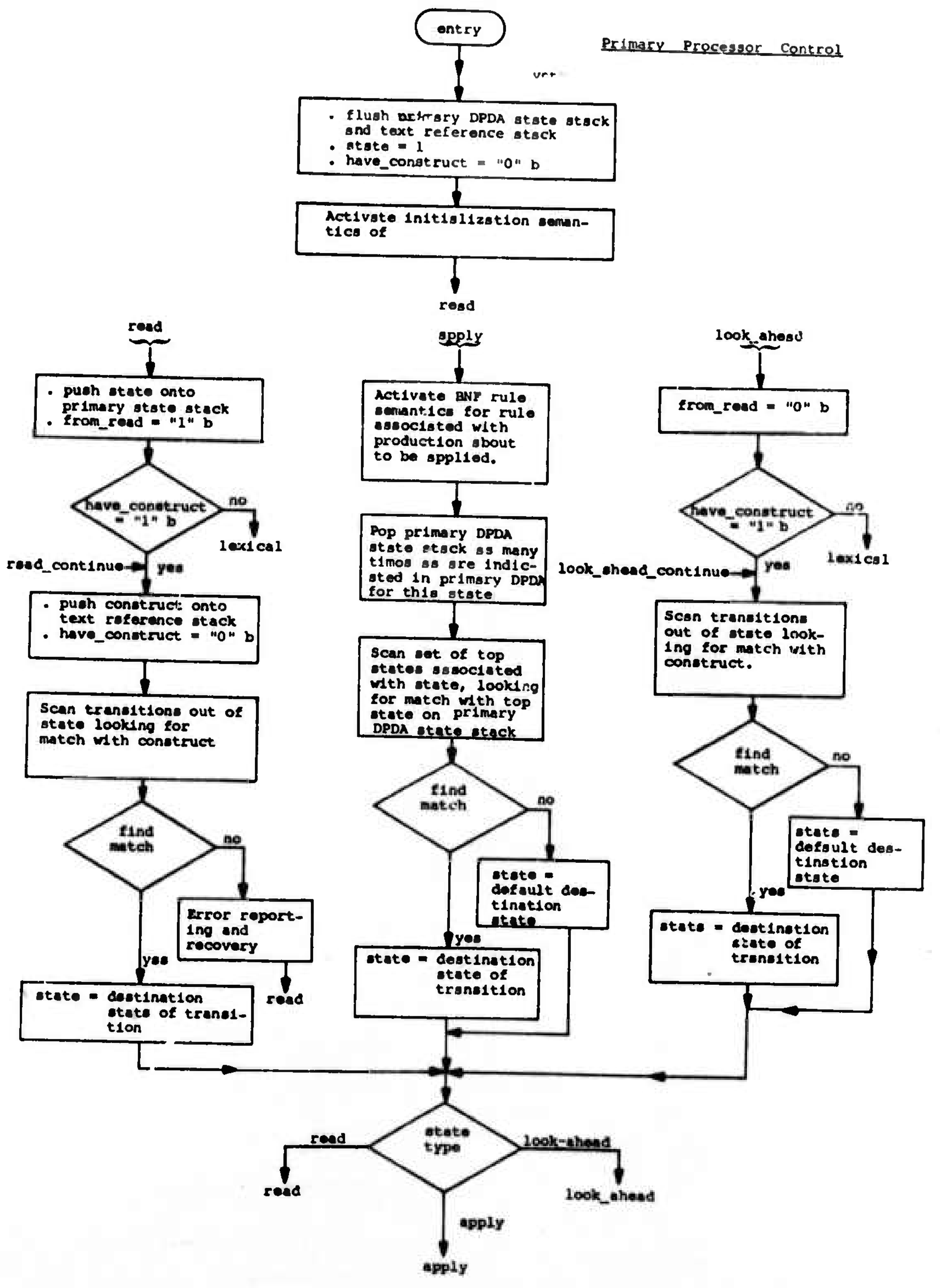
B.2.11 lis_processor_control

lis_processor_control is the procedure that controls the execution of the language processors produced by LIS. That is, it coordinates the lexical parser, the primary parser, and language semantics.

Procedure Size (typical):
   Source: 400 PL/I Statements
   Object: 1000 Words

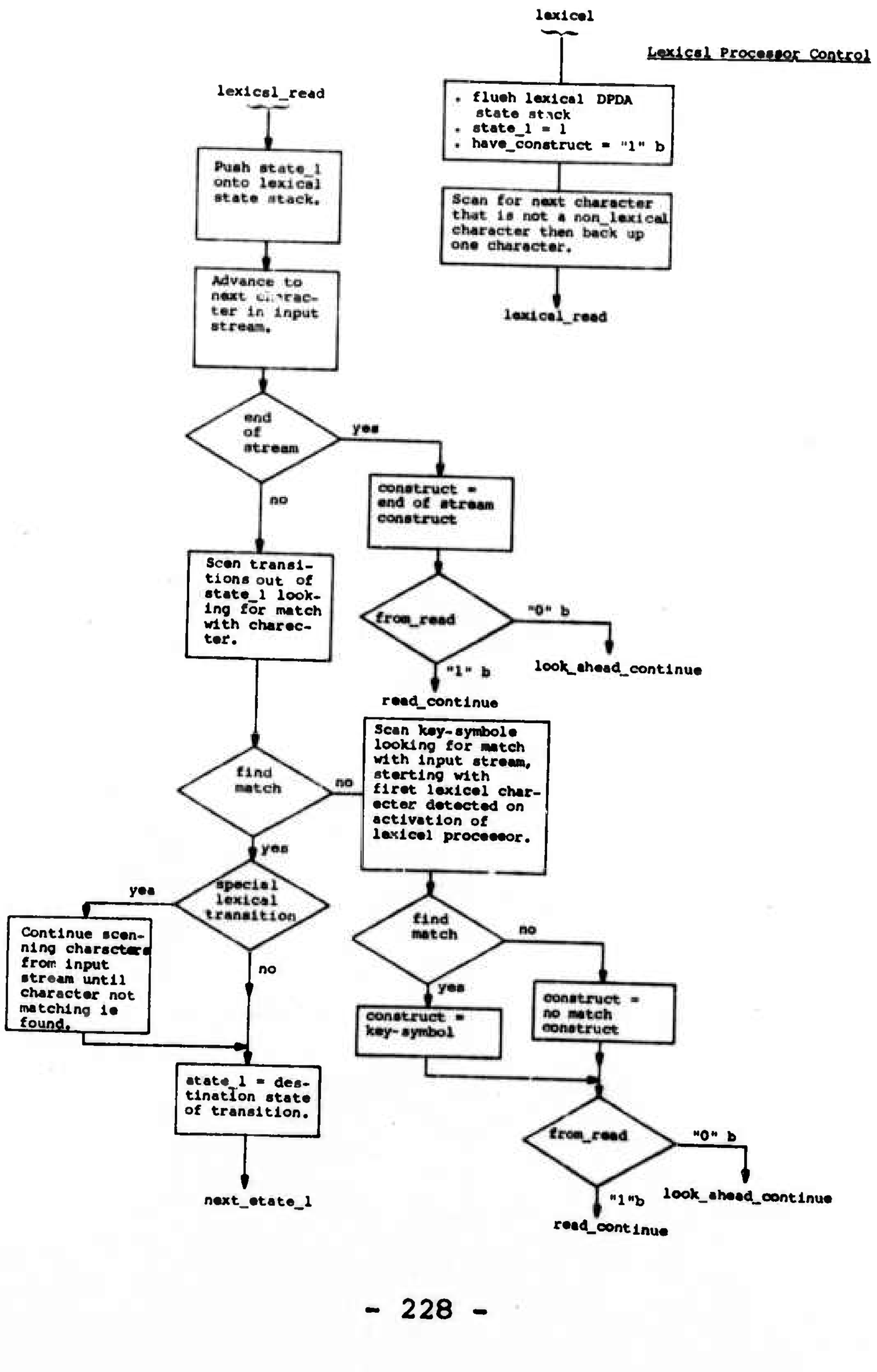
lis_processor_control

Primary Processor Control

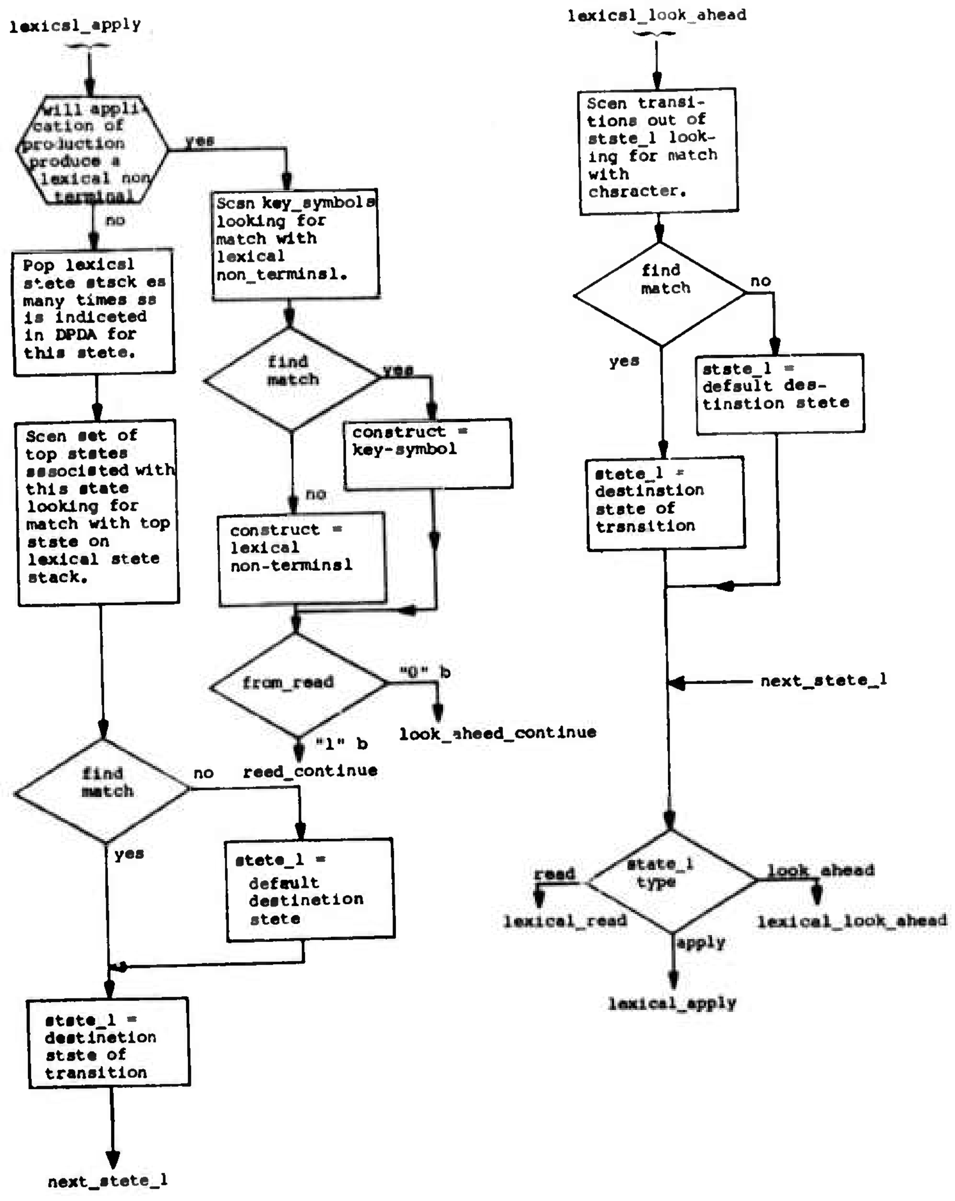


entry
• flush primary DPDA state stack and text reference stack
• state = 1
• have_construct = "0" b
Activate initialization semantics of
read
read
• push state onto primary state stack
• from_read = "1" b
have_construct = "1" b
no
lexical
read_continue
yes
• push construct onto text reference stack
• have_construct = "0" b
Scan transitions out of state looking for match with construct
find match
no
Error reporting and recovery
read
yes
state = destination state of transition
apply
Activate BNF rule semantics for rule associated with production about to be applied.
Pop primary DPDA state stack as many times as are indicated in primary DPDA for this state
Scan set of top states associated with state, looking for match with top state on primary DPDA state stack
find match
no
state = default destination state
yes
state = destination state of transition
look_ahead
from_read = "0" b
have_construct = "1" b
no
lexical
look_ahead_continue
yes
Scan transitions out of state looking for match with construct.
find match
no
state = default destination state
yes
state = destination state of transition
state type
read
read
look-ahead
look_ahead
apply
apply

lis_procassor_control (continued)


lexicel
Lexicel Processor Control
. flush lexical DPDA state stack
. state_l = 1
. have_construct = "1" b
Scan for next character that is not a non_lexical character then back up one character.
lexical_read
lexicsl_read
Push state_l onto lexical state stack.
Advance to next character in input stream.
end of stream
yes
no
construct = end of stream construct
from_read
"0" b
"1" b
look_ahead_continue
read_continue
Scan transitions out of state_l looking for match with character.
find match
no
yes
Scan key-symbols looking for match with input stream, starting with first lexical character detected on activation of lexical processor.
special lexical transition
yes
no
Continue scanning characters from input stream until character not matching is found.
find match
no
yes
construct = key-symbol
construct = no match construct
state_l = destination state of transition.
next_state_l
from_read
"0" b
"1"b
look_ahead_continue
read_continue

lis_processor_control (continued)



lexicsl_apply
will application of production produce a lexical non terminal
yes
no
Pop lexicsl stete stack es many times ss is indiceted in DPDA for this stete.
Scen set of top stetes ssociated with this stste looking for match with top stste on lexicsl stete stack.
find match
no
yes
stete_l = default destinetion stete
stste_l = destinetion stste of transition
next_stete_l
Scsn key_symbols looking for match with lexical non_terminsl.
find match
yes
no
construct = key-symbol
construct = lexical non-terminsl
from_read
"0" b
"1" b
look_shead_continue
reed_continue
lexicsl_look_ahead
Scen transitions out of stste_l looking for match with chsracter.
find match
no
yes
stste_l = defsult destinstion stete
stete_l = destinstion stste of trsnsition
next_stete_l
stste_l type
read
look ahead
apply
lexical_read
lexical_look_ahead
lexical_apply

B.3 Major System Data Structures

In the following discussion, we present a description of the major data structures of the Language Implementation System. In addition, we indicate the maximum storage requirements (in 36 bit words) of these structures during the computation of the primary DPDA of the PL/I grammar given in Appendix D.

The chart on the following page indicates the usage of the major data structures throughout LIS.

Preceding page blank

Procedure \ Structures	grammar	non-terminal	key-symbol	configuration info	configuration	configuration access	CFSM	state access	look-back	initial DPDA	look-ahead	final DPDA	Multics DPDA
lis													
validate_definition	C	C											
analyze_grammar	M	C											
initialize_generator	M	U	C	C									
compute_cfsm	U	U	U	U	C	C	C	C	C				
convert_cfsm_to_dpda	U	U					U		C	C			
look_ahead		U					U			C	C		
optimize_dpda	U	U	U				U			U		C	
multics_dpda_optimization												U	C
lis_debug_monitor	U	U	U	U		U	U	U		U			

Key: C - Created M - Modified U - Used

B.3.1 Grammar Structures: grammar_rules grammar_alts grammar_symbols

The grammar structures exist so as to provide a convenient representation of the grammar submitted in the LIS Language Definition. They are created by validate_definition, modified by analyze_grammar and initialize_generator, and used by compute_cfsm, convert_cfsm_to_dpda, optimize_dpda, and lis_debug_monitor.

grammar_rules contains information on the submitted grammar that is pertinent at the BNF rule level. Entries are made sequentially according to the order of occurrence of the BNF rule in the submitted grammar. The grammar contains a total of n_grammar_rules BNF rules.

defined_non_terminal
: The non-terminal that is defined in the current BNF rule.

n_alts
: The number of alternatives in the current BNF rule.

first_alt_index
: The index into grammar_alts of the first alternative of the current BNF rule.

semantics
: A bit indicating whether semantics is associated with the current BNF rule:
  "0"b -> no semantics
  "1"b -> semantics

grammar_type
: A bit indicating the type of grammar to which the rule belongs:

"0"b -> lexical
"1"b -> primary

grammar_alts contains information on each alternative of the BNF rule. Entries are made sequentially according to the order of occurrence of the alternative in the submitted grammar. The grammar contains a total of n_grammar_alts alternatives.

first_symbol_index
The index into grammar_symbols of the first symbol of the current alternative.

n_symbols
The number of symbols in the current alternative.

any_non_terminals
A bit indicating whether the current alternative contains any non-terminals:
"0"b -> no non-terminals
"1"b -> at least one non-terminal

needed_by_language
A bit indicating whether the current alternative contributes to the language defined by the submitted grammar:
"0"b -> not needed
"1"b -> needed
Set by analyze grammar.

alt_configuration_index
The index into basic_configuration_info of the configuration information associated with the start of the current alternative. Set by initialize_generator.

grammar_symbols describes the symbols that make up the alternatives of the grammar. Entries are made sequentially according to the order of occurrence of the symbol in the submitted grammar. The grammar contains n_grammar_symbols symbols.

symbol
: If the current symbol is non-terminal, then symbol contains the non-terminal code. If the current symbol is a terminal string, then symbol is an index into the LIS Language Definition segment of the start of the terminal string.

l_symbol
: If the current symbol is a non-terminal, then l_symbol equals zero. If the current symbol is a terminal string then l_symbol is the number of characters in the LIS Language Definition segment that make up the terminal string (i.e. escapes have not been resolved).

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 4067 words.

```
1         grammar_rules(n_grammar_rules)               based(grammar_rules_ptr),
          2         defined_non_terminal               fixed     binary(35),
          2         n_alts                             fixed     binary(35),
          2         first_alt_index                    fixed     binary(35),
          2         semantics                          bit(1)    aligned,
          2         grammar_type                       bit(1)    aligned,
n_grammar_rules                                        fixed     binary(35),
grammar_rules_ptr                                      pointer,

1         grammar_alts(n_grammar_alts)                 based(grammar_alts_ptr),
          2         first_symbol_index                 fixed     binary(35),
          2         n_symbols                          fixed     binary(35),
          2         any_non_terminals                  bit(1)    aligned,
          2         needed_by_language                 bit(1)    aligned,
          2         alt_configuration_index            fixed     binary(35),
n_grammar_alts                                         fixed     binary(35),
grammar_alts_ptr                                       pointer,

1         grammar_symbols(n_grammar_symbols)           based(grammar_symbols_ptr),
          2         symbol                             fixed     binary(35),
          2         l_symbol                           fixed     binary(35),
n_grammar_symbols                                      fixed     binary(35),
grammar_symbols_ptr                                    pointer,
```

B.3.2 Non-Terminal Structures: non_terminal_struct non_terminal_cross_refs

The non-terminal structures exist so as to provide a convenient representation of the non-terminals of the submitted grammar. non_terminal_struct is created in validate_definition and non_terminal_cross_refs is created in analyze_grammar. The structures are used in initialize_generator, compute_cfsm, convert_cfsm_to_dpda, look_ahead, optimize_dpda, and lis_debug_monitor.

non_terminal_struct contains a separate entry for each unique ("escapes" resolved) non-terminal of the grammar. Subsequent to the creation of this structure, non-terminals may be referenced by the index of their structure entry. There are n_non_terminals non-terminals in the grammar.

non_terminal_name
The spelling of the non-terminal, with "escapes" resolved.

n_definitions
The number of BNF rules in which the current non-terminal is defined; set by analyze_grammar.

first_definition_index
The index into non_terminal_cross_refs of the first BNF rule in which the current non-terminal is defined; set by analyze_grammar.

n_references
The number of alternatives in which the current non-terminal is referenced; set by analyze_grammar.

first_reference_index
The index into non_terminal_cross_refs of the first alternative in which the current non-terminal is referenced; set by analyze_grammar.

lexical_non_terminal
A bit indicating whether the current non-terminal is a lexical non-terminal:

```
"0"b -> non-terminal not <lexical_non_terminal>
"1"b -> non-terminal is  <lexical_non_terminal>
```

non_terminal_cross_refs is a structure of non-terminal cross references. Entries are sequential in the sense that all definitions or references for a particular non-terminal are contiguous within the structure.

cr_rule
The rule in which the non-terminal is defined/referenced.

cr_alt
If the table entry represents a definition, then cr_alt equals zero. If the entry represents a reference then cr_alt equals the alternative of cr_rule in which the non-terminal is referenced.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 4383 words.

```
1         non_terminal_struct(n_non_terminals)     based(non_terminal_struct_ptr),
          2         non_terminal_name              char(73),
          2         n_definitions                  fixed     binary(35),
          2         first_definition_index         fixed     binary(35),
          2         n_references                   fixed     binary(35),
          2         first_reference_index          fixed     binary(35),
          2         lexical_non_terminal           bit(1)    aligned,
n_non_terminals                                    fixed     binary(35),
non_terminal_struct_ptr                            pointer,

1         non_terminal_cross_refs(n_non_terminal_cross_refs)
                                                   based(non_terminal_cross_refs_ptr),
          2         cr_rule                        fixed     binary(35),
          2         cr_alt                         fixed     binary(35),
n_non_terminal_cross_refs                          fixed     binary(35),
non_terminal_cross_refs_ptr                        pointer,
```

B.3.3 Key-Symbol Structures: key_symbols_struct
key_symbols

The key-symbol structures exist so as to provide a convenient representation of the key-symbols (terminal symbols) of a primary grammar. The structures are created by initialize_generator and used by compute_cfsm, optimize_dpda, and lis_debug_monitor.

key_symbols_struct contains a separate entry for each unique ("escapes" resolved) key-symbol in the primary grammar. Subsequent to the creation of this table, key-symbols may be referenced by the index of their structure entry. There are n_key_symbols key-symbols in the primary grammar.

key_start
: An index into key_symbols indicating the location of the start of the current key-symbol.

key_length
: The number of characters in the current key-symbol.

key_symbols is the character string which is a concatenation of all unique key-symbols in the primary grammar. There are n_key_chars characters in the string.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 473 words.

```
1          key_symbols_struct(n_key_symbols)       based(key_symbols_struct_ptr),
           2          key_start                    fixed      binary(35),
           2          key_length                   fixed      binary(35),
n_key_symbols                                      fixed      binary(35),
key_symbols_struct_ptr                             pointer,

key_symbols                                  char(n_key_chars)   based(key_symbols_ptr),
n_key_chars                                  fixed      binary(35),
key_symbols_ptr                              pointer,
```

B.3.4 Configuration Information Structures:
basic_configuration_info
apply_configuration_info

During the computation of the CFSM, it is necessary to know the correspondence between bit positions in the configurations and the grammar from which the configurations are derived. Thus, one mapping exists from the grammar to the configurations, and another from the configurations to the grammar. The first mapping is provided by grammar_alts.alt_configuration_index. The second mapping is provided by the configuration information structures. The structures are created by initialize_generator, and used by compute_cfsm and lis_debug_monitor.

basic configuration info contains information for each symbol of each alternative of the grammar for which the parser is to be computed. The structure has one entry for each bit position in the basic portion of the configuration, so that the i-th entry corresponds to the i-th position in the configuration.

b_config_rule
The BNF rule to which the configuration bit corresponds.

b_config_alt
The alternative within b_config_rule to which the configuration bit corresponds.

b_config_symbol
The symbol to which the configuration bit corresponds. If the grammar in question is

primary, then the symbol may be the number of a non-terminal or may be the number of a key-symbol, as indicated by b_config_symbol_type. If the grammar in question is lexical, then the symbol may be the number of a non-terminal, may be a terminal character, or may be the index into a temporary table in which are stored the special lexical encodings, as indicated by b_config_symbol_type.

b_config_symbol_type
If the grammar in question is primary, then b_config_symbol is:
"00"b -> number of key-symbol
"01"b -> number of non-terminal
"10"b -> not used
"11"b -> not used
If the grammar in question is lexical, then b_config_symbol is:
"00"b -> terminal character
"01"b -> number of non-terminal
"10"b -> not used
"11"b -> special lexical encoding index

b_config_at_start
A bit indicating whether the configuration bit corresponds to the start of an alternative:
"0"b -> not at start of alternative
"1"b -> at start of alternative

b_config_at_end
An integer, the value of which is interpreted as follows:
^=0 The configuration bit corresponds to the last symbol of an alternative, and the value of b_config_at_end is an index indicating the configuration position at which the alternative is applied.

=0 The configuration bit does not correspond to the last symbol of an alternative.

<u>apply_configuration_info</u> contains information on the application of each alternative of the grammar for which the

parser is to be computed. The structure has one entry for each bit position of the apply portion of the configuration, so that the i-th entry corresponds to the i-th position in the configuration.

a_config_rule
: The BNF rule to which the configuration bit corresponds.

a_config_alt
: The alternative within a_config_rule to which the configuration bit corresponds.

a_config_n_to_pop
: The number of states that will be popped from the DPDA state stack when the alternative is applied.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 5883 words.

```
1         basic_configuration_info(l_basic_configuration)
               based(configuration_info_ptr),
          2         b_config_rule                    fixed     binary(35),
          2         b_config_alt                     fixed     binary(35),
          2         b_config_symbol                  fixed     binary(35),
          2         b_config_symbol_type             bit(2)    aligned,
          2         b_config_at_start                bit(1)    aligned,
          2         b_config_at_end                  fixed     binary(35),
l_basic_configuration                                fixed     binary(35),
configuration_info_ptr                               pointer,

1         apply_config_info(l_basic_configuration + 1: l_configuration)
               based(addr(basic_configuration_info(l_basic_configuration + 1))),
          2         a_config_rule                    fixed     binary(35),
          2         a_config_alt                     fixed     binary(35),
          2         a_config_n_to_pop                fixed     binary(35),
l_configuration                                      fixed     binary(35),
```

B.3.5 Configuration Structures

As discussed in Chapter III, Section III.C, the configurations play an integral part in the computation of a grammar's CFSM. state being generated from, and thus corresponding to, a single configuration. The configuration is, in effect, a computational notation for recording the state configuration of a grammar while computing its CFSM. The configurations consist of two parts, the basic configuration and the apply configuration. The interpretation of these is given in the discussion of the configuration information structures (Section B.3.4). The configurations are created by compute_cfsm and are not used by the system once that procedure has exited.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 20848 words.

```
configuration(current_configuration)    bit(l_configuration)          unaligned
                                 based(configurations_ptr),
configuration_bit(current_configuration, l_configuration)   bit(1)    unaligned
                                 based(configurations_ptr),
configurations_ptr               pointer,
current_configuration            fixed      binary(35),
```

B.3.6 <u>Configuration Access Structures</u>:

configuration_access_list_struct
configuration_access_list

Each time a new configuration is computed, it must be determined whether the same configuration has already been computed. If so, references to the new configuration may be directed to the previous configuration, and the new one destroyed. Since each CFSM state, and hence each configuration, is accessed by one and only one symbol, the configuration access structures are threaded so that all configurations accessed by the same symbol are on the same list. The configuration access structures are created by compute_cfsm and used by lis_debug_monitor.

<u>configuration_access_list_struct</u> is the structure that bounds the threaded list (configuration_access_list) of configurations that are accessed by the same symbol.

first_configuration_accessed_loc(i, 1)
: The location in the list of the first configuration entry accessed by the i-th key-symbol (if grammar is primary) or i-th terminal character, if grammar is lexical.

first_configuration_accessed_loc(i, 2)
: The location in the list of the first configuration entry accessed by the i-th non-terminal, if the grammar is primary, or by the i-th non-terminal or the i-th special lexical encoding, if the grammar is lexical.

last_configuration_accessed_loc(i, 1)
: The location in the list of the last configuration entry accessed by the i-th key-symbol, if the grammar is primary, or by

the i-th terminal character, if the grammar is lexical.

last_configuration_accessed_loc(i, 2)
The location in the list of the last configuration entry accessed by the i-th non-terminal, if the grammar is primary, or by the i-th non-terminal or the i-th special lexical encoding, if the grammar is lexical.

configuration access list is the threaded list of configuration indicators.

configuration_accessed
The number of the accessed configuration.

next_configuration_accessed_loc
The location in the list of the next configuration entry: equals zero if the current entry is the last in the list.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 1849 words.

```
1         configuration_access_list_struct(n_unique_transition_symbols)
                              based(configuration_access_list_struct_ptr),
                                                         fixed      binary(35),
          2         first_configuration_accessed_loc(2)  fixed      binary(35),
          2         last_configuration_accessed_loc(2)   fixed      binary(35),
n_unique_transition_symbols                              pointer,
configuration_access_list_struct_ptr

1         configuration_access_list(n_configuration_access_list_entries)
                              based(configuration_access_list_ptr),
                                                         fixed      binary(35),
          2         configuration_accessed               fixed      binary(35),
          2         next_configuration_accessed_loc      fixed      binary(35),
n_configuration_access_list_entries                      pointer,
configuration_access_list_ptr
```

B.3.7 <u>CFSM Structures</u>: cfsm_states cfsm_read_transitions cfsm_apply_transitions

The CFSM structures comprise the Characteristic Finite State Machine. The structures are created by compute_cfsm, and are used by convert_cfsm_to_dpda, look_ahead, optimize_dpda, and lis_debug_monitor.

<u>cfsm_states</u> contains information that is relevant to the CFSM at the state level. As the CFSM is computed, n_cfsm_states is advanced until a total number of n_cfsm_states + 1 states exist. The 0-th state exists for purposes of initialization and look-ahead.

cfsm_state_type
A bit string indicating the type of state:
"00"b -> read state
"01"b -> apply state
"10"b -> inadequate state
"11"b -> not used

cfsm_accessing_symbol
The symbol with which the state is accessed. If the grammar is primary, then the symbol may be the number of a non-terminal, or may be the number of a key-symbol, as indicated by cfsm_accessing_symbol_type. If the grammar is lexical, then the symbol may be the number of a non-terminal, may be a terminal character, or may be an index into a temporary table in which are stored the special lexical encodings, as indicated by cfsm_accessing_symbol_type.

cfsm_accessing_symbol_type
A bit string indicating the type of symbol accessing the state. If the grammar is primary:
"00"b -> number of key-symbol

```
        "01"b -> number of non-terminal
        "10"b -> not used
        "11"b -> not used
    If the grammar is lexical:
        "00"b -> terminal character
        "01"b -> number of non-terminal
        "10"b -> special lexical encoding index
        "11"b -> not used
```

cfsm_corresponding_dpda_state
The number of the DPDA state which will be generated from the current state. That is, the number of a DPDA read state, DPDA apply state, or DPDA look-ahead state.

cfsm_n_transitions
The number of transitions out of the current state.

cfsm_transitions_ptr
A pointer into the state transitions, indicating the first transition out of the current state.

<u>cfsm_transitions</u>: cfsm_read_transitions
cfsm_apply_transitions

cfsm_read_transitions and cfsm_apply_transitions contain, respectively, information on the read transitions and apply transitions of the CFSM states. As may be seen in their declarations, however, they are not independent structures, rather they both overlay a structure of state transitions, the point of overlay being established by the value of cfsm_transitions_ptr of the state with which the current transitions are associated (n_cfsm_states). The transitions from each state are contiguous in the transitions structure.

__cfsm read_transitions__ overlays the read transitions.

read_transition_symbol_type
  A bit indicating the current transition type.
  If the grammar is primary, then:
    "00"b -> number of key-symbol
    "01"b -> number of non-terminal
    "10"b -> not used
    "11"b -> not used
  If the grammar is lexical, then:
    "00"b -> terminal character
    "01"b -> number of non-terminal
    "10"b -> not used
    "11"b -> special lexical encoding

read_transition_symbol
  The current transition symbol. If the grammar is primary, then the symbol may be the number of a non-terminal, or may be the number of a key-symbol, as indicated by read_transition_symbol_type. If the grammar is lexical, then the symbol may be the number of a non-terminal, may be a terminal character, or may be an index into a temporary table in which are stored the special lexical encodings, as indicated by read_transition_symbol_type.

read_transition_destination
  The CFSM state that is the destination of the current state.

read_transition_dpda_state
  The DPDA state in which the current transition will be found. If the current read transition is from a CFSM read state, then read_transition_dpda_state will be the number of the corresponding DPDA read state. However, if the current transition is from a CFSM inadequate state, then that state will generate a corresponding DPDA look-ahead state, and all read transitions out of the CFSM state will generate a single DPDA read state. In this latter case the value of read_transition_dpda_state will correspond to the number of this generated read state.

read_transition_not_used
  not used.

cfsm apply transitions overlays the apply transitions.

apply_transition_type
Equals "01"b, by definition.

apply_transition_rule
The number of the BNF rule that is to be applied.

apply_transition_alt
The number of the alternative of apply_transition_rule that is to be applied.

apply_transition_dpda_state
The DPDA state in which the current transition will be found. If the transition is from an apply state, then apply_transition_dpda_state will be the number of the corresponding DPDA apply state. However, if the current transition is from an inadequate CFSM state, then that state will generate a corresponding DPDA look-ahead state, and each apply transition out of the CFSM state will generate a separate DPDA apply state. In such a case, the value of apply_transition_dpda_state will correspond to the number of this generated apply state.

apply_transition_n_to_pop
The number of states that will be popped from the DPDA state stack when the alternative is applied.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 25367 words.

```
1         cfsm_states(0: n_cfsm_states)          based(cfsm_states_ptr),
          2         cfsm_state_type              bit(2)    aligned,
          2         cfsm_accessing_symbol        fixed     binary(35),
          2         cfsm_accessing_symbol_type   bit(2)    aligned,
          2         cfsm_corresponding_dpda_state fixed    binary(35),
          2         cfsm_n_transitions           fixed     binary(35),
          2         cfsm_transitions_ptr         pointer,
n_cfsm_states                                    fixed     binary(35),
cfsm_states_ptr                                  pointer,

1         cfsm_read_transitions(cfsm_n_transitions(n_cfsm_states))
                                        based(cfsm_transitions_ptr(n_cfsm_states)),
          2         read_transition_symbol_type  bit(2)    aligned,
          2         read_transition_symbol       fixed     binary(35),
          2         read_transition_destination  fixed     binary(35),
          2         read_transition_dpda_state   fixed     binary(35),
          2         read_transition_not_used     fixed     binary(35),

1         cfsm_apply_transitions(cfsm_n_transitions(n_cfsm_states))
                                        based(cfsm_transitions_ptr(n_cfsm_states)),
          2         apply_transition_type        bit(2)    aligned,
          2         apply_transition_rule        fixed     binary(35),
          2         apply_transition_alt         fixed     binary(35),
          2         apply_transition_dpda_state  fixed     binary(35),
          2         apply_transition_n_to_pop    fixed     binary(35),
```

B.3.8 <u>State Access Structures</u>: state_access_list_struct state_access_list

The state access structures contain an entry for each state in the CFSM. For each state, a threaded list is established, the list containing all states that access the state in question on a single transition. The list was useful in debugging the system and plays an important role in the CLR(k) algorithm. Since the <u>LALR(k)</u> algorithm was replaced in favor of a simpler look-ahead scheme, the structures serve no functional purpose in the current system.

<u>state access list struct</u> is the structure that bounds the threaded list (state_access_list) of accessing states.

sal_state_start_loc
: An index into state_access_list indicating the location of the first state in the list.

sal_state_end_loc
: An index into state_access_list indicating the location of the last state in the list.

<u>state access list</u> is the threaded list of accessing states.

sal_state
: The number of the accessing CFSM state.

sal_next_state_loc
: An index into state_access_list indicating the location of the next state in the list. Equals zero if the current entry is the last entry in the list.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 8653 words.

```
1          state_access_list_struct(n_cfsa_states)
                                             based(state_access_list_struct_ptr),
           2         sal_state_start_loc              fixed     binary(35),
           2         sal_state_end_loc                fixed     binary(35),
state_access_list_struct_ptr                          pointer,

1          state_access_list(n_in_state_access_list)
                                             based(state_access_list_ptr),
           2         sal_state                        fixed     binary(35),
           2         sal_next_state_loc               fixed     binary(35),
n_in_state_access_list                                fixed     binary(35),
state_access_list_ptr                                 pointer,
```

B.3.9 Look Back Structures: look_back_states_struct look_back_states

The look back structures contain an entry for each state in the CFSM. For each CFSM state, S, containing apply transitions, tne structures form a list of states, T, satisfying the following condition:

> A path of symbol transitions exists from S to T which matches, symbol for symbol, the alternative applied in exactly one of the apply transitions of S.

The list is used in converting CFSM apply transitions into DPDA apply states. The structures are created in two steps. In compute_cfsm, the first four elements of look_back_states are set. In convert_cfsm_to_dpda, the internal procedure, compute_look_back, is invoked to create look_back_states_struct and to set the last element of look_back_states, resulting in the desired threaded list.

look_back_states_struct is the structure that bounds the threaded list. For a CFSM state containing no apply transition, both elements are zero.

lb_first_top_state_loc
: An index into look_back_states indicating the location of the first state in the list.

lb_last_top_state_loc
: An index into look_back_states indicating the location of the last state in the list.

look_back_states is the threaded list of states.

lb_top_state
The number of a CFSM state satisfying the condition previously established.

lb_rule
The number of the BNF rule associated with the path between T and S.

lb_alt
The number of the alternative of lb_rule associated with the path between T and S.

lb_destination_state
The state that is reached upon taking the transition from lb_top_state under the non-terminal defined in lb_rule.

lb_next_top_state_loc
An index into look_back_states indicating the location of the next state in the list; equals zero if the current entry is the last entry in the list.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 23460 words.

```
1         look_back_states_struct(n_cfsa_states)  based(look_back_states_struct_ptr),
          2         lb_first_top_state_loc          fixed     binary(35),
          2         lb_last_top_state_loc           fixed     binary(35),
look_back_states_struct_ptr                          pointer,

1         look_back_states(n_look_back_states)    based(look_back_states_ptr),
          2         lb_top_state                    fixed     binary(35),
          2         lb_rule                         fixed     binary(35),
          2         lb_alt                          fixed     binary(35),
          2         lb_destination_state            fixed     binary(35),
          2         lb_next_top_state_loc           fixed     binary(35),
n_look_back_states                                   fixed     binary(35),
look_back_states_ptr                                 pointer,
```

B.3.10 Initial DPDA Structures:

initial_dpda_struct
id_read_states
id_apply_states
id_look_ahead_states
id_read_transitions
id_apply_transitions
id_look_ahead_transitions

The initial DPDA structures contain the non-optimized DPDA. Though the currently configured DPDA could actually be used for language recognition, optimizations will eventually be performed on the structures, resulting in significant improvements in space and time efficiency. The initial DPDA structures are created by convert_cfsm_to_dpda and look_ahead, and are used by optimize_dpda and lis_debug_monitor.

initial_dpda_struct contains information on the structure of the DPDA.

id_k_max
The maximum number of look-ahead symbols required for the grammar.

id_n_read_states
The number of read states in the DPDA.

id_n_apply_states
The number of apply states in the DPDA.

id_n_look_ahead_states
The number of look-ahead states in the DPDA.

id_n_read_transitions
The number of read transitions in the DPDA.

id_n_apply_transitions
The number of apply transitions in the DPDA.

id_n_look_ahead_transitions
The number of look-ahead transitions in the DPDA.

id_read_states contains an entry for each read state in the DPDA.

id_read_n_transitions
The number of read transitions associated with the current state.

id_read_first_transition_index
An index into id_read_transitions indicating the location of the first read transition of the current state.

id_apply_states contains an entry for each apply state of the DPDA.

id_apply_rule
The number of the BNF rule that is to be applied.

id_apply_alt
The number of the alternative of id_apply_rule that is to be applied.

id_apply_n_to_pop
The number of states that will be popped from the DPDA state stack when the alternative is applied.

id_apply_n_top_states
The number of top state (look-back state) transitions associated with the current apply state.

id_apply_first_top_state_index
An index into id_apply_transitions indicating the location of the first apply transition of the current state.

id_look_ahead_states contains an entry for each look-ahead state of the DPDA.

id_look_ahead_n_transitions
The number of look-ahead transitions associated with the current state.

id_look_ahead_first_transition_index
An index into id_look_ahead_transitions indicating the location of the first look-ahead transition of the current state

id_read_transitions contains an entry for each read transition of the DPDA (two entries for a special lexical encoding). The transitions for each read state are contiguous within the structure.

id_read_symbol
The current transition symbol. If the grammar is primary, then the symbol is the number of a key-symbol or lexical non-terminal. If the grammar is lexical, then the symbol is a terminal character or a special lexical encoding, as indicated by the value of id_read_destination. A special lexical encoding is encoded as two transitions. For the first transition, the lower bound encoding character is entered as the symbol, and id_read_destination and id_read_destination_type are 0 and "00"b, respectively. For the second transition, the upper bound encoding character is entered, and id_read_destination and id_read_destination_type are set as appropriate for the destination state of the transition.

id_read_destination
The DPDA state that is the destination for the current read transition; equals zero if the first transition of a special lexical encoding pair.

id_read_destination_type
A bit string indicating the type of destination state for the current read transition:

```
"00"b -> read state
"01"b -> apply state
"10"b -> look-ahead state
"11"b -> not used
```

_id_apply_transitions_ contains an entry for each apply transition of the DPDA. The transitions for each apply state are contiguous within the structure.

id_apply_top_state
The top state for the current apply transition.

id_apply_destination
The DPDA state that is the destination for the current apply transition.

id_apply_destination_type
A bit string indicating the type of destination for the current apply transition:

```
"00"b -> read state
"01"b -> apply state
"10"b -> look-ahead state
"11"b -> not used
```

_id_look_ahead_transitions_ contains an entry for each look-ahead transition of the DPDA. The transitions for each look-ahead state are contiguous within the structure.

id_look_ahead_symbols
The look-ahead symbols for the current transition. If the grammar is primary, then the symbols may be key-symbols or lexical non-terminals. If the grammar is lexical, then each symbol is either a terminal character or the number of a special lexical encoding, as indicated by id_look_ahead_special_lexical.

```
id_look_ahead_destination
   The DPDA state that is the destination for the
   current look_ahead transition.

id_look_ahead_destination_type
   A bit   string   indicating   the   type   of
   destination  state  for the current look-ahead
   transition:
      "00"b -> read state
      "01"b -> apply state
      "10"b -> not used
      "11"b -> not used

id_look_ahead_special_lexical(i)
   A bit indicating whether  the  symbol  in  the
   i-th   position   of  the  current  look-ahead
   transition is a special lexical encoding:

      "0"b -> special lexical encoding
      "1"b -> not special lexical encoding
```

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 33044 words.

```
1         initial_dpda_struct                     based(initial_dpda_ptr),
          2         id_k_max                      fixed     binary(35),
          2         id_n_read_states              fixed     binary(35),
          2         id_n_apply_states             fixed     binary(35),
          2         id_n_look_ahead_states        fixed     binary(35),
          2         id_n_read_transitions         fixed     binary(35),
          2         id_n_apply_transitions        fixed     binary(35),
          2         id_n_look_ahead_transitions   fixed     binary(35),
initial_dpda_ptr                                  pointer,

1         id_read_states(id_n_read_states)        based(id_read_states_ptr),
          2         id_read_n_transitions         fixed     binary(35),
          2         id_read_first_transition_index
                                                  fixed     binary(35),
id_read_states_ptr                                pointer,

1         id_apply_states(id_n_apply_states)      based(id_apply_states_ptr),
          2         id_apply_rule                 fixed     binary(35),
          2         id_apply_alt                  fixed     binary(35),
          2         id_apply_n_to_pop             fixed     binary(35),
          2         id_apply_n_top_states         fixed     binary(35),
          2         id_apply_first_top_state_index
                                                  fixed     binary(35),
id_apply_states_ptr                               pointer,

1         id_look_ahead_states(id_n_look_ahead_states)
                                                  based(id_look_ahead_states_ptr),
          2         id_look_ahead_n_transitions   fixed     binary(35),
          2         id_look_ahead_first_transition_index
                                                  fixed     binary(35),
id_look_ahead_states_ptr                          pointer,
1         id_read_transitions(id_n_read_transitions)
                                        based(id_read_transitions_ptr),
          2         id_read_symbol                char(1)   aligned,
          2         id_read_destination           fixed     binary(35),
          2         id_read_destination_type      bit(2)    aligned,
id_read_transitions_ptr                           pointer,

1         id_apply_transitions(id_n_apply_transitions)
                                        based(id_apply_transitions_ptr),
          2         id_apply_top_state            fixed     binary(35),
          2         id_apply_destination          fixed     binary(35),
          2         id_apply_destination_type     bit(2)    aligned,
id_apply_transitions_ptr                          pointer,

1         id_look_ahead_transitions(id_n_look_ahead_transitions)
                                        based(id_look_ahead_transitions_ptr),
          2         id_look_ahead_symbols         char(3)   aligned,
          2         id_look_ahead_destination     fixed     binary(35),
          2         id_look_ahead_destination_type
                                                  bit(2)    aligned,
          2         id_look_ahead_special_lexical(3)
                                                  bit(1)    unaligned,
id_look_ahead_transitions_ptr                     pointer,
```

B.3.11 <u>Look-Ahead Structures</u>: look_ahead_contour_struct generation_contour_struct

The look-ahead structures are created during the process of resolving each inadequate state of the CFSM.

<u>look_ahead_contour_struct</u> is built up one level (a maximum of three) for each depth of look-ahead required.

lac_n_contour_states
The number of states in the current_contour.

lac_state_index
An index into look_ahead_contour indicating the location of the state from which look-ahead is being performed at the current level.

lac_transition_index
An index indicating the transition within the state identified by lac_state_index from which look-ahead is being performed at the current level.

look_ahead_contour
The contour states for the current level.

<u>generation_contour_struct</u> is used to generate a look-ahead contour. First the contour is generated in generation_contour_struct, and then copied into look_ahead_contour at the appropriate level.

generation_participant(i)
A bit indicating whether CFSM state i has been a participant in the generation process.
"0" -> not a participant
"1"b -> a participant

generation_contour
The contour of generation states.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 229 words.

```
1         look_ahead_contour_struct(3)  based(look_ahead_contour_struct_ptr),
          2         lac_n_contour_states                       fixed     binary(35),
          2         lac_state_index                            fixed     binary(35),
          2         lac_transition_index                       fixed     binary(35),
          2         look_ahead_contour(id_n_read_states)       fixed     binary(35),
look_ahead_contour_struct_ptr                                  pointer,

1         generation_contour_struct     based(generation_contour_struct_ptr),
          2         generation_participant(0: saved_n_cfsm_states)
                                                               bit(1)    unaligned,
          2         generation_contour(saved_n_cfsm_states) fixed     binary(35),
generation_contour_struct_ptr                                  pointer,
```

B.3.12 Final DPDA Structures:

```
final_dpda_struct
fd_read_states
fd_apply_states
fd_look_ahead_states
fd_read_transitions
fd_apply_transitions
fd_look_ahead_transitions
fd_key_symbols_struct
fd_key_symbols
```

The final DPDA structures contain the optimized DPDA. In multics_dpda_optimization we further optimize the DPDA by "fine tuning" it for the Multics System. However, the present structures constitute the essential functional output of the system and are therefore the last of the major system data structures to be discussed in detail. Given these structures, it is a straightforward process to implement a procedure to "fine tune" the structures to a particular computing environment.

final_dpda_struct contains information on the structure of the DPDA.

fd_grammar_type
   A bit indicating the type of grammar from which the DPDA was computed:
      "0"b -> lexical
      "1"b -> primary

fd_k_max
   The maximum number of look-ahead symbols required for the grammar.

fd_read_states_offset
   The offset within the DPDA segment of the start of the read states.

fd_apply_states_offset
The offset within the DPDA segment of the start of the apply states.

fd_look_ahead_states_offset
The offset within the DPDA segment of the start of the look-ahead states.

fd_read_transitions_offset
The offset within the DPDA segment of the start of the read transitions.

fd_apply_transitions_offset
The offset within the DPDA segment of the start of the apply transitions.

fd_look_ahead_transitions_offset
The offset within the DPDA segment of the start of the look-ahead transitions.

fd_non_lexical_chars
If the grammar is lexical, then this element contains the <non_lexical> characters for the lexical parser. If the grammar is primary, then this element is null.

fd_n_key_symbols
If the grammar is primary, then this element contains the sum of the number of key-symbols and the number of non-terminals that are <lexical_non_terminal>. If the grammar is lexical, then this element is zero.

fd_key_symbols_struct_offset
If the grammar is primary then this element contains the offset within the DPDA segment of the start of fd_key_symbols_struct. If the grammar is lexical, then this entry is zero.

fd_n_key_chars
If the grammar is primary, then this element contains the number of characters in fd_key_symbols. If the grammar is lexical, then this element is zero.

fd_key_symbols_offset
If the grammar is primary, then this element contains the offset within the DPDA segment of the start of fd_key_symbols.

fd_next_dpda_offset
   If the grammar is primary and if the DPDA segment also contains a parser for the lexical grammar, then this element contains the offset within the DPDA segment of fd_dpda_struct for the lexical DPDA.

<u>fd_read_states</u> contains an entry for each read state of the DPDA.

fd_read_n_transitions
   The number of read transitions associated with the current state.

fd_read_first_transition_index
   An index into fd_read_transitions indicating the location of the first read transition of the current state.

<u>fd_apply_states</u> contains an entry for each apply state of the DPDA.

fd_apply_rule
   The number of the BNF rule that is to be applied.

fd_apply_alt
   The number of the alternative of fd_apply_rule that is to be applied.

fd_apply_n_to_pop
   The number of states that will be popped from the DPDA stack when the alternative is applied.

fd_apply_semantics
   A bit indicating whether semantics is associated with BNF rule fd_apply_rule in the LIS language Definition:
      "0"b -> no semantics
      "1"b -> semantics

fd_n_top_states
   The number of top state (look back state) transitions associated with the current apply

state.

fd_apply_first_top_state_index
An index into fd_apply_transitions indicating the location of the first apply transition (top state transition) of the current state.

<u>fd_look_ahead_states</u> contains an entry for each look_ahead state of the DPDA.

fd_look_ahead_n_transitions
The number of look-ahead transitions associated with the current state.

fd_look_ahead_first_transition_index
An index into fd_look_ahead_transitions indicating the location of the first look-ahead transition of the current state.

<u>fd_read_transitions</u> contains an entry for each transition of the DPDA (two entries for a special lexical encoding). The transitions for each read state are contiguous within the structure.

fd_read_symbol
The current transition symbol. If the grammar is primary, then the symbol is the number of a key-symbol. If the grammar is lexical, then the symbol is a terminal character or a special lexical encoding, as indicated by the value of fd_read_destination. A special lexical encoding is encoded as two transitions. For the first of such transitions, the lower bound encoding character is entered as the symbol, and fd_read_destination and fd_read_destination_type are 0 and "00"b, respectively. For the second of such transitions, the upper bound encoding is entered and fd_read_destination and fd_read_destination_type are set as appropriate for the destination state of the transition.

fd_read_destination
    The DPDA state that is the destination for the current read transition.

fd_read_destination_type
    A bit string indicating the type of destination state for the current read transition:
        "00"b -> read state
        "01"b -> apply state
        "10"b -> look-ahead state
        "11"b -> not used

<u>fd_apply_transitions</u> contains an entry for each apply transition of the DPDA. The transitions for each apply state are contiguous within the structure.

fd_apply_top_state
    The top state for the current apply transition.

fd_apply_destination
    The DPDA state that is the destination for the current apply transition.

fd_apply_destination_type
    A bit string indicating the type of destination state for the current apply transition:
        "00"b -> read state
        "01"b -> apply state
        "10"b -> look-ahead state
        "11"b -> not used

<u>fd_look_ahead_transitions</u> contains an entry for each look-ahead transition of the DPDA (two entries if at least one symbol of the transition is a special lexical encoding). The transitions for each look-ahead state are contiguous within the structure.

fd_look_ahead_symbols
A sequence of symbols indicating the look-ahead symbol for the current transition. If the grammar is primary, then each symbol is the number of a key-symbol. If the grammar is lexical, then the symbols are either all terminal characters, or contain at least one special lexical encoding, as indicated by the values of fd_look_ahead_destination and fd_look_ahead_destination_type In the case that the symbols contain at least one special lexical encoding, two fd_look_ahead_transitions entry are required, the first such entry being indicated by values of 0 and "00"b for fd_look_ahead_destination and fd_look_ahead_destination_type, respectively. For those symbols of such a transition that are not special lexical encoding symbols (i.e. that are terminal characters), the terminal character is entered in the appropriate position in the first entry and the corresponding position in the second entry is set with the blank character. For those symbols of such a transition that are special lexical encodings, the lower bound encoding character is entered in the appropriate position in the first entry and the upper bound encoding character is entered in the corresponding position in the second entry.

fd_look_ahead_destination
The DPDA state that is the destination for the current look-ahead transition.

fd_look_ahead_destination_type
A bit string indicating the type of destination state for the current transition:
"00"b -> read state
"01"b -> apply state
"10"b -> not used
"11"b -> not used

<u>fd_key_symbols_struct</u> contains a separate entry for each unique ("escapes" resolved) key-symbol and lexical non-terminal in the primary grammar.

fd_key_start
  An index into fd_key_symbol indicating the location of the start of the current key-symbol or lexical non-terminal.

fd_key_length
  The absolute value of this entry is the length of the key-symbol or lexical non-terminal. If the value of the entry is greater than zero, the entry is a key-symbol. If the value of the entry is less than zero, the entry is a lexical non-terminal.

fd_key_symbols is the character string which is a concatenation of all key-symbols and lexical non-terminals.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 7400 words.

```
1         final_dpda_struct                                 based(final_dpda_ptr),
   2          fd_grammar_type                               bit(1)      aligned,
   2          fd_k_max                                      fixed       binary(35),
   2          fd_read_states_offset                         fixed       binary(35),
   2          fd_apply_states_offset                        fixed       binary(35),
   2          fd_look_ahead_states_offset                   fixed       binary(35),
   2          fd_read_transitions_offset                    fixed       binary(35),
   2          fd_apply_transitions_offset                   fixed       binary(35),
   2          fd_look_ahead_transitions_offset
                                                            fixed       binary(35),
   2          fd_non_lexical_chars                          char(8)     aligned,
   2          fd_n_key_symbols                              fixed       binary(35),
   2          fd_key_symbols_struct_offset                  fixed       binary(35),
   2          fd_n_key_chars                                fixed       binary(35),
   2          fd_key_symbols_offset                         fixed       binary(35),
   2          fd_next_dpda_offset                           fixed       binary(35),
 final_dpda_ptr                                             pointer,
1          fd_read_states(fd_n_read_states)        based(fd_read_states_ptr),
           2          fd_read_n_transitions        fixed     binary(35),
           2          fd_read_first_transition_index
                                                   fixed     binary(35),
fd_n_read_states                                   fixed     binary(35),
fd_read_states_ptr                                 pointer,

1          fd_apply_states(fd_n_apply_states)      based(fd_apply_states_ptr),
           2          fd_apply_rule                fixed     binary(35),
           2          fd_apply_alt                 fixed     binary(35),
           2          fd_apply_n_to_pop            fixed     binary(35),
           2          fd_apply_semantics           bit(1)    aligned,
           2          fd_apply_n_top_states        fixed     binary(35),
           2          fd_apply_first_top_state_index
                                                   fixed     binary(35),
fd_n_apply_states                                  fixed     binary(35),
fd_apply_states_ptr                                pointer,

1          fd_look_ahead_states(fd_n_look_ahead_states)
                                                   based(fd_look_ahead_states_ptr),
           2          fd_look_ahead_n_transitions  fixed     binary(35),
           2          fd_look_ahead_first_transition_index
                                                   fixed     binary(35),
fd_n_look_ahead_states                             fixed     binary(35),
fd_look_ahead_states_ptr                           pointer,
1         fd_read_transitions(fd_n_read_transitions)
                                               based(fd_read_transitions_ptr),
                                                       char(1)    aligned,
          2         fd_read_symbol                     fixed      binary(35),
          2         fd_read_destination                bit(2)     aligned,
          2         fd_read_destination_type           fixed      binary(35),
fd_n_read_transitions                                  pointer,
fd_read_transitions_ptr

1         fd_apply_transitions(fd_n_apply_transitions)
                                               based(fd_apply_transitions_ptr),
                                                       fixed      binary(35),
          2         fd_apply_top_state                 fixed      binary(35),
          2         fd_apply_destination               bit(2)     aligned,
          2         fd_apply_destination_type          fixed      binary(35),
fd_n_apply_transitions                                 pointer,
fd_apply_transitions_ptr

1         fd_look_ahead_transitions(fd_n_look_ahead_transitions)
                                               based(fd_look_ahead_transitions_ptr),
                                                       char(fd_k_max)      aligned,
          2         fd_look_ahead_symbols              fixed      binary(35),
          2         fd_look_ahead_destination
          2         fd_look_ahead_destination_type
                                                       bit(2)     aligned,
                                                       fixed      binary(35),
fd_n_look_ahead_transitions                            pointer,
fd_look_ahead_transitions_ptr
```

```
1         fd_key_symbols_struct(fd_n_key_symbols)
                                            based(fd_key_symbols_struct_ptr),
          2         fd_key_start                      fixed     binary(35),
          2         fd_key_length                     fixed     binary(35),
fd_key_symbols_struct_ptr                             pointer,

fd_key_symbols                char(fd_n_key_chars) based(fd_key_symbols_ptr),
fd_key_symbols_ptr            pointer,
```

B.3.13 Multics DPDA Structures

The Multics DPDA structures are an optimized version of the final DPDA structures which are "fine tuned" for the Multics system. As such, they represent no functional advancement over the final DPDA structures and therefore are not discussed in detail. The type of fine tuning done for the Multics environment made a significant impact on both time and space efficiency, and analogous optimizations would be appropriate for other computing environments.

Maximum storage requirements of structures during computation of DPDA of PL/I primary grammar: 1717 words.

```
1         multics_optimized_dpda_syntax_struct                     aligned
                          based(multics_optimized_dpda_syntax_struct_ptr),
          2         mods_grammar_type                              bit(1)    aligned,
          2         mods_k_max                                     bit(18)   unaligned,
          2         mods_read_states_offset                        bit(18)   unaligned,
          2         mods_apply_states_offset                       bit(18)   unaligned,
          2         mods_look_ahead_states_offset                  bit(18)   unaligned,
          2         mods_read_transitions_offset                   bit(18)   unaligned,
          2         mods_apply_transitions_offset                  bit(18)   unaligned,
          2         mods_look_ahead_transitions_offset             bit(18)   unaligned,
          2         mods_n_transition_symbols                      bit(18)   unaligned,
          2         mods_transition_symbols_offset                 bit(18)   unaligned,
          2         mods_n_key_symbols                             bit(18)   unaligned,
          2         mods_key_symbols_struct_offset                 bit(18)   unaligned,
          2         mods_n_key_chars                               bit(18)   unaligned,
          2         mods_key_symbols_offset                        bit(18)   unaligned,
          2         mods_lexical_dpda_offset                       bit(18)   unaligned,
multics_optimized_dpda_syntax_struct_ptr                           pointer,
1         mods_read_states(mods_n_read_states)                /*  1 word  */
                                               based(mods_read_states_ptr),
          2         mods_read_n_transitions                     bit(6)     unaligned,
          2         mods_read_first_transition_index            bit(15)    unaligned,
          2         mods_read_first_transition_symbol_index     bit(15)    unaligned,
mods_n_read_states                                              fixed      binary(35),
mods_read_states_ptr                                            pointer,

1         mods_apply_states(mods_n_apply_states)              /*  1 word  */
                                               based(mods_apply_states_ptr),
          2         mods_apply_rule                             bit(9)     unaligned,
          2         mods_apply_alt                              bit(6)     unaligned,
          2         mods_apply_n_to_pop                         bit(4)     unaligned,
          2         mods_apply_semantics                        bit(1)     unaligned,
          2         mods_apply_n_top_states                     bit(6)     unaligned,
          2         mods_apply_first_top_state_index            bit(10)    unaligned,
mods_n_apply_states                                             fixed      binary(35),
mods_apply_states_ptr                                           pointer,

1         mods_look_ahead_states(mods_n_look_ahead_states)    /*  1 word  */
                                               based(mods_look_ahead_states_ptr),
          2         mods_look_ahead_n_transitions               bit(6)     unaligned,
          2         mods_look_ahead_first_transition_index      bit(15)    unaligned,
          2         mods_look_ahead_first_transition_symbol_index
                                                                bit(15)    unaligned,
mods_n_look_ahead_states                                        fixed      binary(35),
mods_look_ahead_states_ptr                                      pointer,
1         mods_read_transitions(mods_n_read_transitions)          /*  1/2 word  */
                                      based(mods_read_transitions_ptr),
          2         mods_read_destination                  bit(16)   unaligned,
          2         mods_read_destination_type             bit(2)    unaligned,
mods_n_read_transitions                                    fixed     binary(35),
mods_read_transitions_ptr                                  pointer,

1         mods_apply_transitions(mods_n_apply_transitions)        /*  1 word  */
                                      based(mods_apply_transitions_ptr),
          2         mods_apply_top_state                   bit(17)   unaligned,
          2         mods_apply_destination                 bit(17)   unaligned,
          2         mods_apply_destination_type            bit(2)    unaligned,
mods_n_apply_transitions                                   fixed     binary(35),
mods_apply_transitions_ptr                                 pointer,

1         mods_look_ahead_transitions(mods_n_look_ahead_transitions) /*  1/2 word  */
                                      based(mods_look_ahead_transitions_ptr),
          2         mods_look_ahead_destination            bit(16)   unaligned,
          2         mods_look_ahead_destination_type       bit(2)    unaligned,
mods_n_look_ahead_transitions                              fixed     binary(35),
mods_look_ahead_transitions_ptr                            pointer,
mods_transition_symbols        char(mods_n_transition_symbols)      unaligned
                               based(mods_transition_symbols_ptr),

mods_transition_symbols_ptr    pointer,


1          mods_key_symbols_struct(mods_n_key_symbols)         aligned
                                     based(mods_key_symbols_struct_ptr),
           2         mods_key_start                            fixed      binary(35),
           2         mods_key_length                           fixed      binary(35),
mods_key_symbols_struct_ptr                                    pointer,


mods_key_symbols               char(mods_n_key_chars)          unaligned
                               based(mods_key_symbols_ptr),

mods_key_symbols_ptr           pointer,
1         multics_optimized_dpda_lexical_struct                    aligned
                         based(multics_optimized_dpda_lexical_struct_ptr),
          2         modl_grammar_type                              bit(1)    aligned,
          2         modl_k_max                                     bit(18)   unaligned,
          2         modl_read_states_offset                        bit(18)   unaligned,
          2         modl_apply_states_offset                       bit(18)   unaligned,
          2         modl_look_ahead_states_offset                  bit(18)   unaligned,
          2         modl_read_transitions_offset                   bit(18)   unaligned,
          2         modl_apply_transitions_offset                  bit(18)   unaligned,
          2         modl_look_ahead_transitions_offset             bit(18)   unaligned,
          2         modl_non_lexical_characters                    char(8)   aligned,
multics_optimized_dpda_lexical_struct_ptr                          pointer,
1         modl_read_states(modl_n_read_states)                         /*  1/2 word  */
                                                 based(modl_read_states_ptr),
          2         modl_read_n_transitions                    bit(6)     unaligned,
          2         modl_read_first_transition_index           bit(12)    unaligned,
modl_n_read_states                                             fixed      binary(35),
modl_read_states_ptr                                           pointer,

1         modl_apply_states(modl_n_apply_states)                       /*  1 word  */
                                                 based(modl_apply_states_ptr),
          2         modl_apply_rule                            bit(9)     unaligned,
          2         modl_apply_alt                             bit(6)     unaligned,
          2         modl_apply_n_to_pop                        bit(4)     unaligned,
          2         modl_apply_semantics                       bit(1)     unaligned,
          2         modl_apply_n_top_states                    bit(6)     unaligned,
          2         modl_apply_first_top_state_index           bit(10)    unaligned,
modl_n_apply_states                                            fixed      binary(35),
modl_apply_states_ptr                                          pointer,

1         modl_look_ahead_states(modl_n_look_ahead_states)             /*  1/2 word  */
                                                 based(modl_look_ahead_states_ptr),
          2         modl_look_ahead_n_transitions              bit(6)     unaligned,
          2         modl_look_ahead_first_transition_index     bit(12)    unaligned,
modl_n_look_ahead_states                                       fixed      binary(35),
modl_look_ahead_states_ptr                                     pointer,
1         modl_read_transitions(modl_n_read_transitions)          /*  1/2 word  */
                                  based(modl_read_transitions_ptr),
          2         modl_read_symbol                          bit(9)     unaligned,
          2         modl_read_destination                     bit(7)     unaligned,
          2         modl_read_destination_type                bit(2)     unaligned,
modl_n_read_transitions                                       fixed      binary(35),
modl_read_transitions_ptr                                     pointer,

1         modl_apply_transitions(modl_n_apply_transitions)        /*  1 word  */
                                  based(modl_apply_transitions_ptr),
          2         modl_apply_top_state                      bit(17)    unaligned,
          2         modl_apply_destination                    bit(17)    unaligned,
          2         modl_apply_destination_type               bit(2)     unaligned,
modl_n_apply_transitions                                      fixed      binary(35),
modl_apply_transitions_ptr                                    pointer,

1         modl_look_ahead_transitions(modl_n_look_ahead_transitions) /*  1/2 word  */
                                  based(modl_look_ahead_transitions_ptr),
          2         modl_look_ahead_symbol                    bit(9)     unaligned,
          2         modl_look_ahead_destination               bit(7)     unaligned,
          2         modl_look_ahead_destination_type          bit(2)     unaligned,
modl_n_look_ahead_transitions                                 fixed      binary(35),
modl_look_ahead_transitions_ptr                               pointer,
```

Appendix C

LIS Application: p16535

C.1 Introduction

The Common Base Language is being designed by the Computation Structures Group of MIT's Project MAC to serve as the intermediate target representation language (abstract language) of a particular formal semantic system. According to Dennis (Den 72):

> When the meaning of algorithms expressed in some programming language has been specified in precise terms, we say that a formal semantics for the language has been given. A formal semantics for a programming language generally takes the form of two sets of rules -- one set being a translator, and the second set being an interpreter. The translator specifies a transformation of any well-formed program expressed in the source language (the concrete language) into an equivalent program expressed in a second language - the abstract language of the definition. The interpreter expresses the meaning of programs in the abstract language by giving explicit directions for carrying out the computation of any well-formed abstract program as a countable set of primitive steps.

In this appendix, we discuss the design and implementation of a translator from a simple block structured language into the Common Base Language. The presentation assumes a familiarity with the Base Language

(Den 72), though in Part C.3 we define those Base Language Primitives that are used in the translation.

In adding to the theoretical development of the Base Language, the real contribution made by the present effort is probably that of formally specifying the translation of common higher level language constructs into the Base Language primitives, and not necessarily the development of the translator itself. However, that if the Base Language is ever to escape the realm of pure theory and penetrate the world of actual programming language development and implementation, the translator from a particular concrete language into the Base Language will certainly have to be among the first of priorities. Thus our attitude is roughly:

> Given that the Base Language is in a very early stage of development, and even though much progress both in hardware and in software must take place before it will enjoy reasonable acceptance as an implementation device, can we nevertheless do something meaningful towards an implementation given what has been done.

Assuming that a primitive Base Language interpreter will eventually be implement on Multics, the translator presented here can be extended and combined with the interpreter so as to realize a complete translator implementation for a legitimate programming language.

C.2 The pl6535 Language

The concrete language that we implemented is called pl6535. pl6535 is a simple block structured language similar to the language specified by Flinker (Fli 72). However, Flinker's language defines ambiguous expressions, and herein lies the primary difference between his language and pl6535. The BNF specification of the syntax of pl6535 is given below.

The Syntax of pl6535

```
(1)   <primary_non_terminal>  ::=
                              <procedure>  !
(2)   <procedure>  ::=
                              <procedure_head> <body>
                              <procedure_end>  !
(3)   <procedure_head>  ::=
                              <identifier> : PROCEDURE
                              ( <variable_list> ) ;  ¦
                              <identifier> : PROCEDURE ;  !
(4)   <variable_list>  ::=
                              <variable_list> , <identifier>  ¦
                              <identifier>  !
(5)   <body>  ::=
                              <body> <statement>  ¦
                              <statement>  !
(6)   <procedure_end>  ::=
                              END <identifier> ;  !
```

Statements

```
(7)   <statement>  ::=
                              <label> <statement>  ¦
                              <assignment_statement>  ¦
                              <conditional_statement>  ¦
                              <return_statement>  :
                              <goto_statement>  :
                              <declare_statement>  :
                              <procedure>  !
(8)    <label>  ::=
                              <identifier> :  !
(9)    <assignment_statement>  ::=
                              <identifier> = <identifier>
                              ( <variable_list> ) ;  :
                              <identifier> = <expression> ;  !
(10)   <expression>  ::=
                              <expression> + <term>  :
                              <expression> - <term>  :
                              <term>  !
(11)   <term>   ::=
                              <term> * <factor>  :
                              <term> / <factor>  :
                              <factor>  !
(12)   <factor>  ::=
                              ( <expression> )  :
                              <identifier>  :
                              <integer>  !
(13)   <conditional_statement>  ::=
                              IF <equality> THEN <statement>  !
(14)   <equality>  ::=
                              <expression> = <expression>  !
(15)   <return_statement>  ::=
                              RETURN ( <identifier> ) ;  !
(16)   <goto_statement>  ::=
                              GOTO <identifier> ;  !
(17)   <declare_statement>  ::=
                              DECLARE ( <variable_list> ) ;  !
```

<u>Lexical Constructs</u>

```
(18)   <lexical_non_terminal>  ::=
                              <identifier>  :
                              <integer>  !
(19)   <identifier>  ::=
                              <identifier> a->z  :
                              <identifier> A->Z  :
                              <identifier> 0->9  :
                              a->z  :
                              A->Z  !
(20)   <integer>  ::=
                              <integer> 0->9  :
                              0->9  !
```

The first BNF rule specifies that the goal symbol of the primary grammar, <primary_non_terminal>, is defined to be a <procedure>. BNF rule 18 defines <identifier> and <integer> to be <lexical_non_terminal>s, so that LIS will generate a separate parser (the lexical parser) for recognizing these constructs.

In addition to the context-free restrictions placed on the language by the BNF specification, we have the additional (context sensitive) restrictions:

a. No external <procedure> calls are allowed, with the exception of recursion.

b. Non-local goto's are not implemented.

c. All <identifier>s must be declared or defined as follows:

   i. Variables which occur in <expression>s, as arguments in <procedure> calls (BNF (9:1)), or in <return_statement>s, must be explicitly declared in a <declare_statement> or implicitly declared as parameters in a <procedure> definition (BNF (3:1)).

   ii. All <identifier>s referenced by <goto_statement>s must be defined by their occurrence as a <label> in the same block.

   iii. All <procedure>s referenced in a given block must be defined in that block or in a lexicographically enclosing block.

d. The terminal symbols of the primary grammar
   (key-symbols) are reserved.

C.3 The Common Base Language Primitives

The set of Base Language primitives adopted is substantially that defined by Dennis (Den 72) and used by Flinker (Fli 72). Noting that Flinker used a single-address form of the delete instruction while Dennis used a two-address form, we decided that both forms should be accepted, since both are functionally attractive, and since their usage is unambiguous. Two new primitives have been added to solve specific problems: ifgoto to allow conditional transfer, and assign to allow straightforward translation of assignment_statement>s such as "a=b;".

The full list of primitives used is as follows:

assign a,b
    The value of "a" is copied and the copy becomes the value of "b".

const p,q
    Construct an elementary object called "q" having as its value the constant "p".

create p
    Create an elementary object called "p" having no value.

delete p
    The selector "p" no longer exists in the local structure, and all parts of the object which do not share will also cease to exist.

delete p,n
    Like delete p, except that only the "n" branch of the structure "p" is deleted.

goto p
    Take the instruction selected by "p" as the

next to be executed.

ifgoto a,b,p
    If "a" = "b" then goto "p".

link a,b,x
    The "b" branch of "a" is set to share the object at "x".

move p,q
    The program structure "p" is linked to the current local structure under the name "q".

apply p,q
    The instruction following the apply is made dormant, pending the return from the called program "p". A local structure is created for "p", containing a link to the argument structure "q". The next instruction will be the zero-th of "p".

return
    The local structure for the current program is deleted and control returns to the calling program.

select a,b,x
    "x" is set to share the object at the "b" branch of "a".

(add, sub, mult, divide) a,b,x
    Perform "a" op "b" and store the result in "x".

In all of these instructions, should the target structure or object not already exist, it will be created. Readers wishing more graphic explanations of these primitive instructions are referred to the papers by Dennis (Den 72) and Flinker (Fli 72).

C.4 <u>The Structure of the pl6535 Translator</u>

The Language Implementation System was used to implement both the lexical and the primary parsers of the pl6535 translator. The semantics of the translation was specified using PL/I, and the translator was implemented as a two pass compiler. The first pass determines the environment requirements for the <procedure>s of a submitted pl6535 program, while the second pass uses this environment information in generating the actual Base Language translation of that program. The Pass-1 and Pass-2 semantics are specified in detail in Sections C.5 and C.6, respectively.

Due to the current absence of error recovery procedures on LIS, the only pl6535 programs submitted for translation were error free programs.

C.5 The pl6535 Translator: Pass-1

In the literature to date on the Base Language, the approach taken in handling non-local variable references in block structured languages has been to pass these non-local variables as arguments to those procedures in which they are referenced. This may involve several levels of passing, and in attempting to translate such languages, one has two basic choices:

a. Implement a one pass translator, and generate code "on the fly". This involves the chaining of all procedures referencing a particular variable. Then, when the appropriate declaration for that variable has been encountered, its chain is traversed, and the appropriate link and select instructions are inserted in the code already generated.

b. Implement a two pass translator, and use the first pass to determine the environment requirements of each procedure. This means that for each procedure, it will be determined during the first pass which non-local variables must be passed to that procedure when it is called. The second pass then uses this information to generate code in such a way that chaining and insertion of instructions is avoided.

The second of the above methods is by far the simpler, and is the one that we have chosen in our implementation. It avoids the complex chaining strategy associated with the first method, though as will be seen when Pass-2 is discussed, a simplified version of the chaining strategy has been retained for the treatment of <label>s.

C.5.1 Pass-1 Data Structures

Several data structures are used by Pass-1 in generating the <procedure> environments.

text reference stack

```
1    text_reference_stack(top)
                                    based(text_reference_stack_ptr),
     2     construct                       char(10)  unaligned,
top                                        fixed     binary(17),
text_reference_stack_ptr                   pointer,
```

text_reference_stack is a stack containing the lexical constructs recognized by the lexical parser. top marks the top of the stack, and construct contains the actual spelling of the lexical construct for a particular stack entry. The reader should refer to Appendix A for a description of how the text_reference_stack is used to gain access to the lexical constructs.

symbol table

```
1    symbol_table(10),
     2    proc_name_st                 char(10),
     2    symbol_count                 fixed      binary(17),
     2    symbol_entry(20),
          3    symbol_spell            char(10),
          3    declared                bit(1),
current_level                          fixed      binary(17),
```

Our implementation allows a nesting of a maximum of 10 block levels, the current level being indicated by current_level. During the parse of a pl6535 program, and during the execution of that program, only one block associated with each level may be active at a given instant. symbol_table represents the symbol table for a maximum of 10 blocks that are simultaneously active, and identifies the active blocks.

Within symbol_table, the entries have the following meaning.

proc_name_st
    The name of the <procedure> for which the symbol table has been established.

symbol_count
    The number of symbols entered into the table for proc_name_st.

symbol_entry
    Each entry in the symbol table for proc_name_st has two parts:

symbol_spell
    The symbol name.

declared
    A bit indicating whether symbol_spell is declared in proc_name_st:
        "0"b -> not declared.
        "1"b -> declared.

In the semantics for Pass-1, we have implemented a procedure for making entries into the symbol_table. When a non-declaration entry is made in the symbol_table:

> The symbol table for current_level is scanned to see if the entry has already been made. If so, the procedure returns. If not, the entry is made and its declared bit is set to "0"b.

When a symbol declaration is made in the symbol_table:

> The symbol table for current_level is scanned to see if the entry has already been made. If so, the procedure insures that the declared bit of the entry is set to "1"b. If the entry has not been made, the procedure makes the entry and sets its declared bit to "1"b.

<u>var_list</u>

```
var_list(20)          char(10),
var_count             fixed       binary(17),
```

var_list is used to build up the names of <identifier>s that occur in <variable_list>s. The number of entries in var_list is indicated by var_count.

<u>environment_needed_tree</u>

```
1    environment_needed_tree(10),
     2     proc_name_env      char(10),
     2     env_count          fixed      binary(17),
     2     env_entry(20)      char(10),
proc_count                    fixed      binary(17),
```

This is the output from Pass-1. Our implementation allows a maximum of ten uniquely named <procedure>s in any pl6535 program, and the environment_needed_tree exists so as to indicate the environment requirements of each <procedure>. The entries have the following meaning.

proc_name_env
The name of the <procedure> for which the environment information has been determined.

env_count
    The number of variables that constitute the environment requirements of proc_name_env.

env_entry
    The actual names of the variables that constitute the environment requirements of proc_name_env.

proc_count
    The number of <procedure>s in the p16535 program being translated.

C.5.2 Pass-1 Semantics

In the following discussion, we present the basic semantic actions required to determine the environment requirements. The discussion is relative to the BNF definition of pl6535, and any BNF rule not mentioned has no semantic action during this pass. The listings at the end of the appendix should be referenced for details of the implementation.

<procedure_head> (BNF-3)

Upon detecting a <procedure_head>:

a. An environment_needed_tree entry is established for the <procedure>.

b. The <procedure> name is entered into the current_level as being defined (unless this is the first <procedure>, in which case current_level = 0).

c. current_level is incremented, and the symbol_table for current_level is initiated on behalf of the <procedure> name.

d. The parameters of the <procedure> are entered into the symbol_table as having been declared.

<variable_list> (BNF-4)

Detecting a <variable_list> results in var_list being built up to contain the variables (<identifier>s>) in the list.

<procedure_end> (BNF-6)

Upon detecting a <procedure_end>:

a. The symbol_table for current_level is scanned to determine if any symbols are undeclared at this level. If undeclared symbols exist, they are entered into the environment_needed list of the <procedure> being ended and are also entered in the symbol table of the lexicographically enclosing <procedure> (unless current_level = 1).

b. current_level is decremented by 1.

<assignment_statement> (BNF-9)

Upon detecting an <assignment_statement>, we enter the symbol on the left side of the equal sign into the symbol table. In the case of the <procedure> call (BNF (9:1)), we enter the arguments of the call into the symbol table, as well as into var_list.

<factor> (BNF-12)

Upon detecting a <factor>, we know that we are building up an <expression>. <identifier>s which are <factor>s are entered into the symbol_table by the semantics of this rule.

<return_statement> (BNF-15)

Upon detecting a <return_statement>, we enter the name of the variable being returned into the symbol_table.

<declare_statement> (BNF-17)

Upon detecting a <declare_statement>, we enter the

<identifier>s being declared into the symbol_table as having been declared.

C.6 The pl6535 Translator: Pass-2

It is during Pass-2 that the actual translation of a pl6535 program into its Common Base Language representation takes place.

The structure of the Base Language is such that no variables may be referenced, or <procedure>s called which are not part of the currently executing <procedure>. Thus, external references and calls of a higher level language must be translated into non-external (i.e. local) references and calls of the Base Language. These variables and <procedure>s must be passed down to the referencing <procedure> during the calling sequence in the same manner as arguments. A unique naming convention was adopted for the selectors from the $ARG node (created as the second argument of the apply instruction) and the $PAR node (created in the call of a <procedure>, with the same value as the argument of the apply instruction which invoked this <procedure>). In this convention, arguments to be passed, and formal parameters used, are given consecutive integer selectors beginning with 1; all environment variables and <procedure>s are selected with character strings spelling their old names; and the value to be returned (on the $ARG/$PAR structure) is given the selector "$RET".

C.6.1 Pass-2 Data Structures

Several data structures are used by Pass-2 in generating the actual Common Base Language representation of a pl6535 program.

text reference stack

text_reference_stack is used the same way in Pass-2 as it is in Pass-1, so that its description under Pass-1 also applies here.

environment needed tree

environment_needed_tree is the output from Pass-1, and represents the environment requirements for each <procedure>, which is necessary in the generation of the Common Base Language translation. The discussion of environment_needed_tree in Pass-1 also applies here.

var list

var_list is used the same way in Pass-2 as it is in Pass-1, so that its description under Pass-1 also applies here.

label list

```
1    label_list(10),
       2    label_count              fixed     binary(17),
       2    label(15),
              3    label_spell       char(10)  unaligned,
              3    label_def         bit(1),
              3    label_loc         fixed     binary(17),
        3     label_usage_count  fixed     binary(17),
        3     label_usage(10)    fixed     binary(17),
```

label_list is used to handle the pl6535 <goto_statement>s as well as the Base Language goto's that are generated during translation of the pl6535 <conditional_statement>s. A maximum of 10 <label>s can be defined in any block. label_list is filled in as a <procedure> is parsed, and its fields have the following meaning:

label_count
: The number of <label>s in the current <procedure>.

label
: For each <label> in the <procedure>, the following information is ultimately determined:

label_spell
: The name of the <label>.

label_def
: A bit string indicating whether the <label> has been defined:
  "0"b -> not defined.
  "1"b -> defined.

label_loc
: The relative line number in the current block where the <label> is defined.

label_usage_count
: The number of times that the <label> has been referenced.

label_usage
: The absolute Base code line numbers in which the <label> is referenced.

base_code

```
1    base_code(200),
     2     opcode            char(18)  unaligned,
     2     adr1              char(10)  unaligned,
     2     adr2              char(10)  unaligned,
     2     adr3              char(10)  unaligned,
     2     depth             fixed     binary(17),
     2     line_no           fixed     binary(17),
base_code_index              fixed     binary(17),
```

base_code is built up during Pass-2, and contains the Common Base Language representation of the pl6535 program being translated. base_code_index indicates the number of Base Language instructions, and the fields of base_code have the following meaning:

opcode
: The Common Base Language primitive.

adr1
: The first operand.

adr2
: The second operand.

adr3
: The third operand.

depth
: The depth of the current <procedure>.

line_no
: The line number in the Base code output text at which the translation of the current <procedure> began.

depth and line_no are needed for purposes of printing the Common Base Language translation.

C.6.2 Pass-2 Semantics

In addition to the per-rule semantics about to be described, a set of internal procedures have been implemented in Pass-2 which should make the Pass-2 implementation easier to follow. They appear in the listings at the end of the appendix, and are "commented" so that their functions should be apparent to the reader.

In the following discussion, we present the basic semantic actions required to generate the Common Base Language translation of a pl6535 program. The discussion is relative to the BNF definition of pl6535, and any BNF rule not mentioned has no semantic action during this pass. The listings at the end of the appendix should be referenced for details of the implementation.

<procedure_head>/<procedure_end> (BNF-3/BNF-6)

pl6535 is a block structured language and the Base Language is tree structured. Thus, detecting a <procedure_head> or <procedure_end> must cause a corresponding change in the depth of the Base code. Therefore,when detecting such <statement>s, current_level is changed and the output routine is notified to change the indentation of any Base code generated thereafter. <procedure_head>s also represent the entries into the

<procedure> and thus we must output Base code to select the parameter values (select $PAR,i, name_of_formal_parameter(i); i=1 to number of parameters). We must also determine and output Base code to select the environment needed by the <procedure> (select $PAR, name(i), name(i); where each name represents an element of environment_needed_tree for this <procedure>).

<variable_list> (BNF-4)

The semantics of <variable_list> during Pass-2 is the same as during Pass-1, so that its description under Pass-1 also applies here.

procedure call (BNF (9:1))

Upon detecting a <procedure> call, we must output Base code to create a "$ARG" structure and then link each of the arguments to the structure via integer selectors. If the call is not recursive, then we must output Base code to move the text of the called <procedure> from the procedure structure to the data or local structure (see Den 72 for a description of the procedure, data, and local structures). If the call is recursive, then the text will not be accessible in the <procedure> structure but will have been passed as environment needed on the $PAR structure, and selected during this <procedure>'s invocation. The test to check for recursive calls is to see if the <procedure> to be called is in the calling <procedure>'s environment_needed_tree.

Once the appropriate procedure structure has been accessed, Base code must be generated to apply execution of the procedure structure to its associated argument structure. We must subsequently output Base code instructions to select the returned value from the argument structure and to delete the argument structure.

<return_statement> (BNF-15)

Upon detecting a <return_statement>, we must generate Base code to link the returned variable to the parameter structure, and return to the code following the invoking apply instruction.

<expression>/<assignment_statement>

When any of the binary operators are detected (BNF (10:1), BNF (10:2), BNF (11:1), BNF (11:2)), a Base code instruction specifying the corresponding operation is generated, with its first two arguments being popped off the expression_stack. A unique temporary name is used for the resulting sub-expression, which is then pushed onto the expression_stack. <integer> constants (BNF (12:3)) must be given unique names, created using the const instruction, pushed onto the expression_stack. <identifier>s encountered in <expression>s (BNF (12:2)) are simply pushed onto the expression_stack. <assignment_statement>s (BNF (9:2)) such as "x=a;" must be translated into an assign Base code primitive. However, <assignment_statement>s such as "x=3;" can be translated into a single const primitive involving no temporaries. Thus, the semantics for BNF (9:2) must either generate an assign instruction, with one argument coming from the expression_stack, or perform some code optimization by modifying the previous line of Base code so that the temporary name is never used. The semantics associated with the <assignment_statement> is also responsible for some garbage collection, which amounts to deleting all temporaries created in translating the <expression>.

<conditional statement> (BNF-13)

A conditional <statement> is first recognized by BNF-14 (<equality>) where the ifgoto Base code is generated. The compare operands of ifgoto are taken from the expression_stack and the goto location is the present location plus an offset. This is followed by code to delete all temporaries used in the two <expression>s (in case the condition failed) and a goto with a blank address field. This address field represents the location of the start of the Base code translation of the next <statement>, which is not known until BNF-13 is applied. Thus, the semantics of BNF-13 fills in this address field with the correct location. The jump location of the ifgoto follows, and delete's are generated to perform garbage collection if the jump is taken.

<goto_statement> (BNF-16)

Upon detecting a <goto_statement>, a check is made to see if the destination <label> has been defined. If so, translation is straightforward: a Base language goto instruction with the argument taken from label_list.label_loc. If not, the address field of the Base code goto instruction must be left blank and the location of the instruction entered into the label_list. As <label>s are defined (BNF-8), the value of the <label> (the next base_code instruction-selector number) is entered into the label_list, and any existing references (i.e., goto instructions with blank argument fields) are then filled in using the information previously stored in the label_list for that <label>.

C.7 Examples of p16535 Programs and Their Translation

At the end of the appendix are examples of two p16535 programs and their translation. Program p1 is taken directly from Flinker. Program p5 was written to test all of the features of the language, especially those that were not treated in previous works. In the paper by Altman, Gearing, and Weekly (AGW 72), a manual interpretation is given for a portion of p5. Our emphasis in this Appendix being on the p16535 translator, we have omitted the interpretation.

C.8 Conclusions

In implementing a "useful" programming language, such as Algol or PL/I, the following issues will have to be faced, some by the Common Base Language theory, and some by our translator:

The handling of non-local goto's.

Optimization of the Common Base Language code.

Resolving of external function calls without creating cycles.

Defining the means of interaction with the Base Language Machine - operating system, input/output.

Parallelism in computation and its implication for language translation and interpretation.

Generalized data structures, arrays, etc.

Symbol manipulation, character handling, and general data type conversion.

Of course, the implementation of any language, "useful" or not, will have to wait on the implementation of a Common Base Language interpreter. And so, even though the Common Base Language is still in its infancy, perhaps the development of a primitive interpreter is one of the next steps that should be taken.

```
/*
                The_PLe232_Language  -  Pass 1
*/

/*       The Text Reference Stack        */

         dcl      1        text_reference_stack(top)     based(text_reference_stack_ptr),
                           2         construct           char(10)  unaligned,
                  top                                    fixed     binary(17)          external,
                  text_reference_stack_ptr               pointer   external,


/*       The output from pass 1        */

                  1        environment_needed_tree(10)             external,
                           2         proc_name_env                 char(10),
                           2         env_count                     fixed     binary(17),
                           2         env_entry(20)                 char(10),
                  proc_count                                       fixed     binary(17)          external,


/*       The symbol table used in constructing the environment tree.        */

                  1        symbol_table(10)                        static,
                           2         proc_name_st                  char(10),
                           2         symbol_count                  fixed     binary(17),
                           2         symbol_entry(20),
                                     3         symbol_spell        char(10),
                                     3         declared            bit(1),
                  current_level                                    fixed     binary(17)          static,


                  i                  fixed     binary(17)          static,
                  j                  fixed     binary(17)          static,
                  pass_1_error       bit(1)    external,
                  tree_index         fixed     binary(17)          static,
                  var_count          fixed     binary(17)          static,
                  var_list(20)       char(10)  static;

         proc_count = 0;
         current_level = 0;
         pass_1_error = "0"b;
         return;


enter_symbol:   proc(symbol, declare_action);   /*                          *     enter_symbol     *
                                                                            ************************
         This procedure makes entries into the symbol table on a
         per procedure basis.  */
```

```
dcl        declare_action        bit(1)     aligned,
           i                     fixed      binary(17),
           symbol                char(10);


do i = 1 to symbol_count(current_level);
       if symbol = symbol_spell(current_level, i)
          then do;
                  if declare_action
                     then declared(current_level, i) = "1"b;
                  return;
          end;
end;

symbol_count(current_level) = symbol_count(current_level) + 1;
symbol_spell(current_level, symbol_count(current_level)) = symbol;
declared(current_level, symbol_count(current_level)) = declare_action;
return;
end;


                                                                       *     windup     *
windup:    entry(pl6535_program);   /*                                 ******************

           This procedure prints the envi-onment requirements.   */


           dcl        ioa_                  entry,
                      ioa_$nnl              entry,
                      pl6535_program        char(*);


           call ioa_("^|^2-The Environment Requirements For: ^a.pl6535^2/",
                      pl6535_program);
           call ioa_("Procedure^-Environment Needed");
           do i = 1 to proc_count;
                      call ioa_$nnl("^/^a^-", proc_name_env(i));
                      do j = 1 to env_count(i);
                                 call ioa_$nnl("^a, ", env_entry(i, j));
                      end;
           end;
           call ioa_("^/");
           return;
```

```
<primary_non_terminal>  ::=
                                  <procedure>  !

/*        no semantics          */
          return;


<procedure>  ::=
                                  <procedure_head> <body> <procedure_end>  !

/*        no semantics          */
          return;


<procedure_head>  ::=
                                  <identifier> : PROCEDURE ( <variable_list> ) ;  !
                                  <identifier> : PROCEDURE ;  !

          if alternative_number = 1
             then i= 6;
             else i = 3;
          proc_count = proc_count + 1;
          proc_name_env(proc_count) = construct(top - i);
          env_count(proc_count) = 0;
          if current_level ^= 0
             then call enter_symbol(construct(top - i), "1"b);
          current_level = current_level + 1;
          proc_name_st(current_level) = construct(top - i);
          symbol_count(current_level) = 0;
          do i = 1 to var_count;
                 call enter_symbol(var_list(i), "1"b);
          end;
          return;


<variable_list>  ::=
                                  <variable_list> , <identifier>  !
                                  <identifier>  !

          if alternative_number = 1
             then var_count = var_count + 1;
             else var_count = 1;
          var_list(var_count) = construct(top);
          return;


<body>  ::=
                                  <body> <statement>  !
                                  <statement>  !

/*        no semantics          */
          return;
```

```
<procedure_end>  ::=
                              END <identifier> ;  !

          var_count = 0;
          do i = 1 to proc_count;
                  if proc_name_st(current_level) = proc_name_env(i)
                     then tree_index = i;
          end;

          do i = 1 to symbol_count(current_level);
                  if ^declared(current_level, i)
                     then do;
                              env_count(tree_index) = env_count(tree_index) + 1;
                              env_entry(tree_index, env_count(tree_index)) = symbol_spell(current_level, i);
                              var_count = var_count + 1;
                              var_list(var_count) = symbol_spell(current_level, i);
                     end;
          end;

          current_level = current_level - 1;
          do i = 1 to var_count;
                  if current_level = 0
                     then do;
                              call ioa_("Error! ^a is referenced, but not declared.", var_list(i));
                              pass_1_error = "1"b;
                     end;
                  else call enter_symbol(var_list(i), "0"b);
          end;
          return;
```

```
/*
                    Statements
*/


<statement>  ::=
                              <label> <statement>  |
                              <assignment_statement>  |
                              <conditional_statement>  |
                              <return_statement>  |
                              <goto_statement>  |
                              <declare_statement>  |
                              <procedure>  |

/*       no semantics         */
         return;


<label>  ::=
                              <identifier> :  |

/*       no semantics         */
         return;


<assignment_statement>  ::=
                              <identifier> = <identifier> ( <variable_list> ) ;  |
                              <identifier> = <expression> ;  |

         if alternative_number = 1
            then do;
                   call enter_symbol(construct(top - 6), "0"b);
                   do i = 1 to var_count;
                             call enter_symbol(var_list(i), "0"b);
                   end;
                   call enter_symbol(construct(top - 4), "0"b);
            end;
            else call enter_symbol(construct(top - 3), "0"b);
         return;


<expression>  ::=
                              <expression> + <term>  |
                              <expression> - <term>  |
                              <term>  |

/*       no semantics         */
         return;


<term>   ::=
                              <term> * <factor>  |
                              <term> / <factor>  |
                              <factor>  |
/*        no semantics          */
          return;


<factor>  ::=
                                ( <expression> )  !
                                <identifier>  !
                                <integer>  !

          if alternative_number = 2
             then call enter_symbol(construct(top), "0"b);
          return;


<conditional_statement>  ::=
                                IF <equality> THEN <statement>  !

/*        no semantics          */
          return;


<equality>  ::=
                                <expression> = <expression>  !

/*        no semantics          */
          return;


<return_statement>  ::=
                                RETURN ( <identifier> ) ;  !

          call enter_symbol(construct(top - 2), "0"b);
          return;


<goto_statement>  ::=
                                GOTO <identifier> ;  !

/*        no semantics          */
          return;


<declare_statement>  ::=
                                DECLARE ( <variable_list> ) ;  !

          do i = 1 to var_count;
                   call enter_symbol(var_list(i), "1"b);
          end;
          return;
```

```
/*
                    Lexical Constructs
*/


<lexical_non_terminal>  ::=
                                <identifier>  !
                                <integer>  !

<identifier>  ::=
                                <identifier> a->z  !
                                <identifier> A->Z  !
                                <identifier> 0->9  !
                                a->z  !
                                A->Z  !

<integer>  ::=
                                <integer> 0->9  !
                                0->9  !
```

```
/*
               The_PLo235_Language - Pass 2
*/


/*        The Text Reference Stack          */

dcl       1          text_reference_stack(top)       based(text_reference_stack_ptr),
                     2         construct             char(10)  unaligned,
          top                                        fixed     binary(17)          external,
          text_reference_stack_ptr                   pointer   external;


/*        The output from pass 1        */

dcl       1          environment_needed_tree(10)               external,
                     2         proc_name_env                   char(10),
                     2         env_count                       fixed     binary(17),
                     2         env_entry(20)                   char(10),
          proc_count                                           fixed     binary(17)          external;


/*        The output from pass 2        */

dcl       1          base_code(200)                            external,
                     2         opcode                          char(10)  unaligned,
                     2         adr1                            char(10)  unaligned,
                     2         adr2                            char(10)  unaligned,
                     2         adr3                            char(10)  unaligned,
                     2         depth                           fixed     binary(17),
                     2         line_no                         fixed     binary(17),
          base_code_index                                      fixed     binary(17)          external;


/*        The list of labels within the current procedure, their
          definitions and references.        */

dcl       1          label_list(10)                            static,
                     2         label_count                     fixed     binary(17),
                     2         label(15),
                               3         label_spell           char(10)  unaligned,
                               3         label_def             bit(1),
                               3         label_loc             fixed     binary(17),
                               3         label_usage_count     fixed     binary(17),
                               3         label_usage(10)       fixed     binary(17),


          current_depth                  fixed     binary(17)          static,
          current_level                  fixed     binary(17)          static,
          current_line_no(20)            fixed     binary(17)          static,
          dummy_cn                       char(10)  unaligned static,
```

Preceding page blank

```
        dummy_fb                      fixed     binary(17)          static,
        env_tree_counter              fixed     binary(17)          static,
        env_tree_index(20)            fixed     binary(17)          static,
        expression_stack(100)         char(10)  unaligned static,
        expression_stack_top          fixed     binary(17)          static,
        fb_to_char                    entry(fixed binary(17))       internal  returns(char(13) var),
        forget_temp                   entry     internal,
        goto_stack(10)                fixed     binary(17)          static,
        goto_stack_top                fixed     binary(17)          static,
        i                             fixed     binary(17)          static,
        j                             fixed     binary(17)          static,
        last_adr_of_last_line         entry     internal  returns(char(10) unaligned),
        last_temp                     entry     internal  returns(char(10) unaligned),
        moved                         bit(1)    static,
        n                             fixed     binary(17)          static,
        new_temp                      entry     internal  returns(char(10) unaligned),
        next_base_code_line_no        entry     internal  returns(fixed binary(17)),
        next_base_code_rel_line_no    entry     internal  returns(fixed binary(17)),
        number_of_temps               fixed     binary(17)          static,
        output_base_code              entry(char(18) unaligned, char(10) unaligned, char(10) unaligned,
                                            char(10) unaligned)     internal,
        output_end_code               entry     internal,
        output_proc_code              entry(char(10) unaligned)     internal,
        pop                           entry     internal  returns(char(10) unaligned),
        punct(2)                      char(1)   unaligned,
        push                          entry(char(10) unaligned)     internal,
        put_goto_adr                  entry(fixed binary(17), char(10) unaligned)       internal,
        reset_last_adr_of_last_line   entry(char(10) unaligned)     internal,
        temp_1                        char(10)  unaligned static,
        temp_2                        char(10)  unaligned static,
        temp_opcode                   char(18)  unaligned static,
        var_count                     fixed     binary(17)          static,
        var_list(20)                  char(10)  unaligned static;


    base_code_index = 0;
    current_depth = 0;
    current_level = 0;
    env_tree_counter = 0;
    expression_stack_top = 0;
    number_of_temps = 0;
    return;



windup:  entry(ol6535_program);   /*                                             *     windup     *
                                                                                 ******************
    This entry prints the Common Base Language translation of
    the ol6535 source program.  */


    dcl      ioa_                 entry,
             ioa_$nnl             entry,
             pl6535_program       char(*),
             teos                 char(10)  init((10) "          ");
```

```
call ioa_("^3/^1^2-The Common Base Language Translation of: ^a.p'6535^3/",
        pdp6535_program);

do i = 1 to base_code_index;
        punct(1), punct(2) = ",";
        if adr2(i) = " "
           then punct(1) = " ";
        if adr3(i) = " "
           then punct(2) = " ";
        if line_no(i) >= 0
           then call ioa_("^a^d^-^a^-^a^a ^a^a ^a", substr(tabs, 1, depth(i) + 1), line_no(i),
                        opcode(i), adr1(i), punct(1), adr2(i), punct(2), adr3(i));
        else do;
                dummy_fb = 2;
                if i = 1
                   then dummy_fb = 1;
                depth(i + 1) = 0;
                call ioa_$nnl("^/^a^-^a", substr(tabs, 1, depth(i)), opcode(i));
        end;
end;
call ioa_("^3/");
return;


new_temp:   proc  returns(char(10) unaligned);   /*                              *     new_temp     *
                                                                                 ********************
        This procedure creates a new temporary variable.  */


        dcl       temp                char(10)  unaligned;


        number_of_temps = number_of_temps + 1;
        temp = "$"||fb_to_char(number_of_temps);
        return(temp);
        end;


forget_temp:   proc;   /*                                                        *     forget_temp     *
                                                                                 ***********************
        This procedure forgets that the current temporary variable
        ever existed.  */


        number_of_temps = number_of_temps - 1;
        end;


last_temp:   proc  returns(char(10) unaligned);   /*                             *     last_temp     *
                                                                                 *********************
```

```
          This procedure returns the name of the last temporary
          variable, then forgets that it existed.  */


          dcl       temp                char(10)  unaligned;


          number_of_temps = number_of_temps - 1;
          temp = "$"||fb_to_char(number_of_temps + 1);
          return(temp);
          end;


output_base_code:   proc(opcode_, adr1_, adr2_, adr3_);   /*                  *     output_base_code      *
                                                                                ****************************
          This procedure is used to create a line of base code.  */


          dcl       opcode_             char(10)  unaligned,
                    adr1_               char(10)  unaligned,
                    adr2_               char(10)  unaligned,
                    adr3_               char(10)  unaligned;


          base_code_index = base_code_index + 1;
          current_line_no(current_depth) = current_line_no(current_depth) + 1;
          opcode(base_code_index) = opcode_;
          adr1(base_code_index) = adr1_;
          adr2(base_code_index) = adr2_;
          adr3(base_code_index) = adr3_;
          depth(base_code_index) = current_depth;
          line_no(base_code_index) = current_line_no(current_depth);
          return;
          end;


next_base_code_line_no:   proc  returns(fixed binary(17));   /*              *     next_base_code_line_no     *
                                                                                *********************************
          This procedure returns the absolute line number that will
          be associated with the next line of base code.  */


          return(base_code_index + 1);
          end;


next_base_code_rel_line_no:   proc  returns(fixed binary(17));   /*          *     next_base_code_rel_line_no     *
                                                                                *************************************
          This procedure returns the relative (current_depth) line number
          that will be associated with the next line of base code.  */
          return(current_line_no(current_depth) + 1);
          end;


output_proc_code:   proc(proc_name);   /*                                    *      output_proc_code      *
                                                                             ****************************
          This procedure creates the base code associated with a
          procedure_head.  */


          dcl        proc_name            char(10)  unaligned;


          base_code_index = base_code_index + 1;
          opcode(base_code_index) = proc_name;
          adr1(base_code_index) = " ";
          adr2(base_code_index) = " ";
          adr3(base_code_index) = " ";
          depth(base_code_index) = current_depth;
          line_no(base_code_index) = -1;
          current_depth = current_depth + 1;
          current_line_no(current_depth) = -1;
          return;
          end;


output_end_code:   proc;   /*                                                *      output_end_code      *
                                                                             ***************************
          This procedure creates the base code associated with a
          procedure_end.  */


          current_depth = current_depth - 1;
          return;
          end;


last_adr_of_last_line:   proc  returns(char(10) unaligned);   /*             *      last_adr_of_last_line      *
                                                                             *********************************
          This procedure returns the last non-blank address field of the
          last instruction.  */


          if adr3(base_code_index) ^= " "
             then return(adr3(base_code_index));
             else return(adr2(base_code_index));
          end;
```

```
reset_last_adr_of_last_line:   proc(adr_);   /*                      *     reset_last_adr_of_last_line      *
                                                                     ****************************************
        This procedure sets the last non-blank address field of the
        last instruction equal to adr_.  */


        dcl       adr_                char(10)  unaligned;


        if adr3(base_code_index) ^= " "
           then adr3(base_code_index) = adr_;
           else adr2(base_code_index) = adr_;
        return;
        end;


put_goto_adr:   proc(abs_line_no, adr_);   /*                           *     put_goto_adr     *
                                                                        ************************
        This procedure sets the destination address field of the goto
        instruction at abs_line_no equal to adr_.  */


        dcl       abs_line_no         fixed binary(17),
                  adr_                char(10)  unaligned;


        adr1(abs_line_no) = adr_;
        return;
        end;


push:   proc(top_of_stack);   /*                                        *     push     *
                                                                        ****************
        This procedure pushes the variable, top_of_stack, onto the
        expression stack.  */


        dcl       top_of_stack        char(10)  unaligned;


        expression_stack_top = expression_stack_top + 1;
        expression_stack(expression_stack_top) = top_of_stack;
        return;
        end;


pop:    proc  returns(char(10) unaligned);   /*                         *     pop     *
                                                                        ***************
        This procedure pops the top variable off the expression stack and
        returns it.  */
```

```
          dcl       top_of_stack        char(10)  unaligned;


          top_of_stack = expression_steck(expression_stack_top);
          expression_stack_top = expression_steck_top-1;
          return(top_of_stack);
          end;


fb_to_char:   proc(v)  returns(cher(13) ver);   /*                                 *     fb_to_cher     *
                                                                                    **********************
          This procedure converts a fixed binery nuember into its
          equivelent cnaracter string representation.  */

          dcl                 e         fixed     binary(35),
                              c         cher(4)   based(addr(m)),
                              ii        fixed     binery(35)          init(14),
                              e         fixed     binery(35),
                              neg       bit(1),
                              r         cher(13)  verying,
                              s         cher(13),
                              v         fixed     binery(35);


          neg = "0"b;
          if v = 0
             then return("0");
          if v < 0
             then neg = "1"b;
          e = abs(v);
          do while(a>0);
                    e = eod(e, 10) + 48;
                    ii = ii - 1;
                    suostr(s, ii, 1) = substr(c, 4, 1);
                    e = divide(e, 10, 35, 0);
          end;
          r = substr(s, ii);
          if neg
             then r = "-"||r;
          return(r);
          end;
```

```
<primary_non_terminal>  ::=
                                <procedure>  |

/*          no semantics        */
            return;


<procedure>  ::=
                                <procedure_head> <body> <procedure_end>  ;

/*          no semantics        */
            return;


<procedure_head>  ::=
                                <identifier> : PROCEDURE ( <variable_list> ) ;  |
                                <identifier> : PROCEDURE ;  |

            if alternative_number = 1
               then i = 6;
               else do;
                       i = 3;
                       var_count = 0;
               end;
            current_level = current_level + 1;
            call output_proc_code(construct(top - i));
            do  i = 1 to var_count;
                       call output_base_code("select", "SPAR", fb_to_char(i), var_list(i));
            end;
            env_tree_index(current_level), env_tree_counter = env_tree_counter + 1;  /* pass 1 insures correct order */
            do i = 1 to env_count(env_tree_index(current_level));
                       call output_base_code("select", "SPAR", env_entry(env_tree_index(current_level), i),
                                                               env_entry(env_tree_index(current_level), i));

            end;
            label_count(current_level) = 0;
            return;


<variable_list>  ::=
                                <variable_list> , <identifier>  |
                                <identifier>  |

            if alternative_number = 1
               then var_count = var_count + 1;
               else var_count = 1;
            var_list(var_count) = construct(top);
            return;


<body>  ::=
                                <body> <statement>  |
                                <statement>  ;

/*          no semantics        */
            return;
```

```
<procedure_end>  ::=
                              END <identifier> ;  !

        current_level = current_level - 1;
        call output_end_code;
        return;
```

```
/*                  Statements
*/


<statement>  ::=
                              <label> <statement>  |
                              <assigneent_stateeent>  |
                              <conditional_stateeent>  |
                              <return_stateeent>  |
                              <goto_stateeent>  |
                              <declare_stateeent>  |
                              <procedure>  ;

/*        no seeantics        */
          return;


<label>  ::=
                              <identifier> :  |

          do i = 1 to label_count(current_level);
                    if label_spell(current_level, i) = construct(top - 1)
                       then go to label_found;
          end;
          i, label_count(current_level) = label_count(current_level) + 1;
          label_spell(current_level, i)= construct(top - 1);
          label_usage_count(current_level, i) = 0;
label_found:   label_def(current_level, i) = "1"b;
          label_loc(current_level, i) = next_base_code_rel_line_no;
          do j = 1 to label_usage_count(current_level, i);
                    dueey_fb = next_base_code_rel_line_no;
                    dueey_ch = fb_to_char(dueey_fb);
                    call put_goto_adr(label_usage(current_level, i, j), dueey_ch);
          end;
          return;


<assigneent_stateeent>  ::=
                              <identifier> = <identifier> ( <variable_list> ) ;  |
                              <identifier> = <expression> ;  !

          if alternative_nueber = 1
             then do;
                    call output_base_code("create", "$ARG", " ", " ");
                    do i = 1 to var_count;
                              call output_base_code("link", "$ARG", fb_to_char(i), var_list(i));
                    end;
                    moved = "0"b;
                    do i = 1 to env_count(env_tree_index(current_level));
                              if construct(top - 4) = env_entry(env_tree_index(current_level), i)
                                 then go to no_eove;
                    end;
                    call output_base_code("move", construct(top - 4), construct(top - 4), " ");
                    eoved = "1"b;
no_move:            do n = 1 to proc_count;
```

```
                  if proc_name_env(n) = construct(top - 4)
                     then go to set_links;
                  end;
set_links:        do i = 1 to env_count(n);
                        call output_base_code("link", "$ARG", env_entry(n, i), env_entry(n, i));
                  end;
                  call output_base_code("apply", construct(top - 4), "$ARG", " ");
                  call output_base_code("select", "$ARG", "$RET", construct(top - 6));
                  call output_base_code("delete", "$ARG", " ", " ");
                  if moved
                     then call output_base_code("delete", construct(top - 4), " ", " ");
              end;
              else do;
                  temp_1 = pop;
                  if last_adr_of_last_line = temp_1
                     then do;
                        call reset_last_adr_of_last_line(construct(top - 3));
                        call forget_temp;
                     end;
                     else call output_base_code("assign", temp_1, construct(top - 3), " ");
                  do while(number_of_temps > 0);
                        dummy_ch = last_temp;
                        call output_base_code("delete", dummy_ch, " ", " ");
                  end;
              end;
           return;


<expression>  ::=
                              <expression> + <term>  |
                              <expression> - <term>  |
                              <term>  ;

           if alternative_number = 3
              then return;
           if alternative_number = 1
              then temp_opcode = "add";
              else temp_opcode = "sub";
std_exp:   temp_1 = pop;
           temp_2 = new_temp;
           dummy_ch = pop;
           call output_base_code(temp_opcode, dummy_ch, temp_1, temp_2);
           call push(temp_2);
           return;


<term>  ::=
                              <term> * <factor>  |
                              <term> / <factor>  |
                              <factor>  ;

           if alternative_number = 3
              then return;
           if alternative_number = 1
              then temp_opcode = "mult";
              else temp_opcode = "divide";
           go to std_exp;
```

```
<factor>  ::=
                              ( <expression> )  |
                              <identifier>  |
                              <integer>  ;

          if alternative_number = 1
             then return;
          if alternative_number = 2
             then call push(construct(top));
             else do;
                    temp_1 = new_temp;
                    call output_base_code("const", construct(top), temp_1, " ");
                    call push(temp_1);
             end;
          return;


<conditional_statement>  ::=
                              IF <equality> THEN <statement>  ;

          dummy_fb = next_base_code_rel_line_no;
          dummy_ch = fo_to_char(dummy_fb);
          call put_goto_adr(goto_stack(goto_stack_top), dummy_ch);
          goto_stack_top = goto_stack_top - 1;
          return;


<equality>  ::=
                              <expression> = <expression>  ;

          temp_1 = pop;
          temp_2 = pop;
          dummy_fb = next_base_code_rel_line_no + 2 + number_of_temps;
          dummy_ch = fo_to_char(dummy_fb);
          call output_base_code("ifgoto", temp_2, temp_1, dummy_ch);
          n = number_of_temps;
          do while(number_of_temps > 0);
                    dummy_ch = last_temp;
                    call output_base_code("delete", dummy_ch, " ", " ");
          end;
          goto_stack_top = goto_stack_top + 1;
          goto_stack(goto_stack_top) = next_base_code_line_no;
          call output_base_code("goto", " ", " ", " ");
          number_of_temps = n;
          do while(number_of_temps > 0);
                    dummy_ch = last_temp;
                    call output_base_code("delete", dummy_ch, " ", " ");
          end;
          return;


<return_statement>  ::=
                              RETURN ( <identifier> ) ;  ;
          call output_base_code("link", "$PAR", "$RET", construct(top - 2));
          call output_base_code("return", " ", " ", " ");
          return;


<goto_statement>  ::=
                              GOTO <identifier> ;  !

          do i = 1 to label_count(current_level);
                  if label_spell(current_level, i) = construct(top - 1)
                     then go to used_label;
          end;
          i, label_count(current_level) = label_count(current_level) + 1;
          label_spell(current_level, i) = construct(top - 1);
          label_def(current_level, i) = "0"b;
          label_usage_count(current_level, i) = 0;
used_label:   label_usage_count(current_level, i)=
                label_usage_count(current_level, i) + 1;
          label_usage(current_level, i,
                      label_usage_count(current_level, i)) = next_base_code_line_no;
          if label_def(current_level, i)
             then do;
                  dummy_fb = label_loc(current_level, i);
                  dummy_ch = fb_to_char(dummy_fb);
                  call output_base_code("goto", dummy_ch, " ", " ");
             end;
          else call output_base_code("goto", " ", " ", " ");
          return;


<declare_statement>  ::=
                              DECLARE ( <variable_list> ) ;  !

/*        no semantics         */
          return;
```

```
/*
                    Lexical Constructs
*/


<lexical_non_terminal>  ::=
                          <identifier>  |
                          <integer>  |

<identifier>  ::=
                          <identifier> a->z  |
                          <identifier> A->Z  |
                          <identifier> 0->9  |
                          a->z  |
                          A->Z  |

<integer>  ::=
                          <integer> 0->9  |
                          0->9  |
```

p1.p16535

```
P1:   PROCEDURE;
          DECLARE(a, b, c);
          a = 4;
          b = Q1(a);
          Q1: PROCEDURE(n);
                    DECLARE(m, n, k);
                    m = 1;
                    k = m + n;
                    RETURN(k);
          END Q1;
END P1;
```

The Common Base Language Translation of: p1.p16535

```
P1:   PROCEDURE;
          DECLARE(a, b, c);
          a = 4;
          P1        0         const     4, a

          b = Q1(a);
                    1         create    $ARG
                    2         link      $ARG, 1, a
                    3         move      Q1, Q1
                    4         apply     Q1, $ARG
                    5         select    $ARG, $RET, b
                    6         delete    $ARG
                    7         delete    Q1

Q1:       PROCEDURE(n);
                    DECLARE(m, n, k);
                    Q1        0         select    $PAR, 1, n

                    m = 1;
                              1         const     1, m

                    k = m + n;
                              2         add       m, n, k

                    RETURN(k);
                              3         link      $PAR, $RET, k
                              4         return

          END Q1;
END P1;
```

p5.p16535

```
P5:   PROCEDURE;
            DECLARE (a, b, c, d);
            a = 0;
            b = 2;
            c = Q51(a, b);
            d = Q52(b, c);
Q51:        PROCEDURE(x, y);
                      DECLARE(x, y, z);
                      IF x+1 = y*y  THEN RETURN(y);
                      IF y = 4  THEN IF x = 3  THEN x = 2;
                      x = (x*x) + 1/(y - 5);
                      z = Q51(x, y);
                      RETURN(z);
            END Q51;
            a = d;
Q52:        PROCEDURE(p, q);
                      DECLARE(p, q);
                      IF a = 1  THEN GOTO alpha;
            beta:     RETURN(q);
            alpha:    q = R5(p);
                      GOTO beta;
            R5:       PROCEDURE(x);
                                DECLARE(x, y);
                                y = x*b;
                                RETURN(y);
                      END R5;
            END Q52;
END P5;
```

The Environment Requirements For: p5.p16535

```
Procedure Environment Needed

P5
Q51       Q51,
Q52       a, b,
R5        b,
```

Preceding page blank

The Common Base Language Translation of: p5.p16535

```
P5:   PROCEDURE;
           DECLARE (a, b, c, d);
           a = 0;
           P5         0          const     0, a

           b = 2;
                      1          const     2, b

           c = Q51(a, b);
                      2          create    $ARG
                      3          link      $ARG, 1, a
                      4          link      $ARG, 2, b
                      5          move      Q51, Q51
                      6          link      $ARG, Q51, Q51
                      7          apply     Q51, $ARG
                      8          select    $ARG, $RET, c
                      9          delete    $ARG
                      10         delete    Q51

           d = Q52(b, c);
                      11         create    $ARG
                      12         link      $ARG, 1, b
                      13         link      $ARG, 2, c
                      14         move      Q52, Q52
                      15         link      $ARG, a, a
                      16         link      $ARG, b, b
                      17         apply     Q52, $ARG
                      18         select    $ARG, $RET, d
                      19         delete    $ARG
                      20         delete    Q52


Q51:       PROCEDURE(x, y);
                      DECLARE(x, y, z);
                      Q51        0          select    $PAR, 1, x
                                 1          select    $PAR, 2, y
                                 2          select    $PAR, Q51, Q51

                      IF x+1 = y*y  THEN RETURN(y);
                                 3          const     1, $1
                                 4          add       x, $1, $2
                                 5          mult      y, y, $3
                                 6          ifgoto    $2, $3, 11
                                 7          delete    $3
                                 8          delete    $2
                                 9          delete    $1
                                 10         goto      16
                                 11         delete    $3
                                 12         delete    $2
                                 13         delete    $1
                                 14         link      $PAR, $RET, y
                                 15         return

                      IF y = 4  THEN IF x = 3  THEN x = 2;
                                 16         const     4, $1
```

```
                    17          ifgoto    y, $1, 20
                    18          delete    $1
                    19          goto      27
                    20          delete    $1
                    21          const     3, $1
                    22          ifgoto    x, $1, 25
                    23          delete    $
                    24          goto      27
                    25          delete    $1
                    26          const     2, x

                x = (x*x) + 1/(y - 5);
                    27          mult      x, x, $1
                    28          const     1, $2
                    29          const     5, $3
                    30          sub       y, $3, $4
                    31          divide    $2, $4, $5
                    32          add       $1, $5, x
                    33          delete    $5
                    34          delete    $4
                    35          delete    $3
                    36          delete    $2
                    37          delete    $1

                z = Q51(x, y);
                    38          create    $ARG
                    39          link      $ARG, 1, x
                    40          link      $ARG, 2, y
                    41          link      $ARG, Q51, Q51
                    42          apply     Q51, $ARG
                    43          select    $ARG, $RET, z
                    44          delete    $ARG

                RETURN(z);
                    45          link      $PAR, $RET, z
                    46          return

        END Q51;
        a = d;
                21          assign    d, a


Q52:    PROCEDURE(p, q);
                DECLARE(p, q);
                Q52         0           select    $PAR, 1, p
                            1           select    $PAR, 2, q
                            2           select    $PAR, a, a
                            3           select    $PAR, b, b

                IF a = 1  THEN GOTO alpha;
                            4           const     1, $1
                            5           ifgoto    a, $1, 8
                            6           delete    $1
                            7           goto      10
                            8           delete    $1
                            9           goto      12

        beta:   RETURN(q);
                            10          link      $PAR, $RET, q
                    11        return

        alpha:   q = R5(p);
                    12        create    $ARG
                    13        link      $ARG, 1, p
                    14        move      R5, R5
                    15        link      $ARG, b, b
                    16        apply     R5, $ARG
                    17        select    $ARG, $RET, q
                    18        delete    $ARG
                    19        delete    R5

                 GOTO beta;
                    20        goto      10

        R5:      PROCEDURE(x);
                    DECLARE(x, y);
                    R5        0         select    $PAR, 1, x
                              1         select    $PAR, b, b

                    y = x*b;
                              2         mult      x, b, y

                    RETURN(y);
                              3         link      $PAR, $RET, y
                              4         return

                 END R5;
        END Q52;
    END P5;
```

Appendix D

LIS Application: PL/I

D.1 Introduction

In this appendix, we present the primary and lexical grammar of a large subset of the IBM Laboratory Vienna's specification of the concrete syntax of PL/I (AOU 68).

D.2 PL/I Primary Grammar

The primary grammar of the PL/I subset is given at the end of the appendix. It is a highly inclusive subset of the full Vienna specification, including declarations, input/output, and on-conditions. The grammar is the most complex grammar yet submitted to LIS.

In developing the primary parser for the PL/I subset, it was discovered that the Vienna definition contains at least two areas of syntactic ambiguity. The first area of ambiguity occurs between the definition of labellist and reference, as they may both appear at the beginning of a statement. The Vienna definition of these constructs is indicated below.

```
labellist ::=
        ( ( identifier ; initial-label ) : )...
initial-label ::=
        [( identifier[((, •signed-integer...))].)... ]
        identifier((, •signed-integer...))
        [(.identifier)... ]

reference ::=
        [reference ->] basic-reference
basic-reference ::=
        (. unqualified-reference   )
unqualified-reference ::=
        identifier[((,  (expression ; *)   ))]
```

As an example of the ambiguity inherent in these definitions, consider the following partial phrase at the beginning of a statement:

susie(1, 2, 3, 4, 5, 6, 7, 8

Is the partial phrase the beginning of an initial-label or the beginning of a reference on the left side of an assignment statement? The illustrated ambiguity cannot be resolved with finite look-ahead, and the Vienna definition is therefore not LR(k). To circumvent this problem, we defined <labellist> as shown in the LIS Language Definition at the end of the appendix. Our definition admits "illegal" labels, which are easily detected by semantics.

The second area of ambiguity is more obvious and involves the following definition of datalist:

```
datalist ::=
        (, •datalist-element...)
datalist-element ::=
        (datalist DO do-specification) ; expression
```

Our remedy for this ambiguity is also indicated in the LIS Language Definition at the end of the appendix.

Once the ambiguities were corrected, LIS was able to produce a highly efficient parser for the PL/I subset (Chapter III, Section III.C.2). This application illustrates the use of LIS both as a language development tool and as an implementation facility. As a language development tool, the system identified areas of syntactic ambiguity, which are difficult for both programmer and machine. As an implementation facility, it may be noted that the author was able to implement the parsers (lexical and primary) for the PL/I subset in less than one week. This implementation time would have been significantly less, had it not been for the clerical task of entering the PL/I grammar into a Multics segment.

D.3 PL/I Lexical Grammar

The PL/I lexical grammar given in the LIS Language Definition at the end of the appendix is the most comprehensive lexical grammar yet submitted to LIS. It includes all of the Vienna lexical constructs except for sterling-constant, picture-specification, and comment. Comment is more appropriately implemented by hand in LIS Processor Control.

```
/*        The  Primary  Grammar  of  PL/I          */

<primary_non_terminal>  ::=
                              <program>  !


/*        Program  Structure          */

<program>  ::=
                              <procedure_list>  !

<procedure_list>  ::=
                              <procedure>  !
                              <procedure_list> <procedure>  !

<procedure>  ::=
                              <procedure_1>  !
                              <prefixlist> <procedure_1>  !

<procedure_1>  ::=
                              <procedure_header> <body>  !

<procedure_header>  ::=
                              <labellist> procedure ;  !
                              <labellist> procedure <parameterlist> ;  !
                              <labellist> procedure <procedure_optionslist> ;  !
                              <labellist> procedure <parameterlist> <procedure_optionslist> ;  !

<parameterlist>  ::=
                              ( <id_comma_list> )  !

<id_comma_list>  ::=
                              <identifier>  !
                              <id_comma_list> , <identifier>  !

<procedure_optionslist>  ::=
                              <procedure_option>  !
                              <procedure_optionslist> , <procedure_option>  !

<procedure_option>  ::=
                              returns ( <function_attribute_list> )  !
                              recursive  !
```

Preceding page blank

```
<function_attribute_list>  ::=
                               <function_attribute>  |
                               <function_attribute_list> <function_attribute>  ;

<body>  ::=
                               <end_clause>  |
                               <sentence_list> <end_clause>  ;

<sentence_list>  ::=
                               <sentence>  |
                               <sentence_list> <sentence>  ;

<sentence>  ::=
                               <statement>  |
                               <procedure>  |
                               <entry>  |
                               <declaration_sentence>  |
                               <format_sentence>  |

<end_clause>  ::=
                               <end_clause_1>  |
                               <labellist> <end_clause_1>  |

<end_clause_1>  ::=
                               end <identifier> ( <integer> ) ;  |
                               end <identifier> ;  |
                               end ;  ;

<entry>  ::=
                               <labellist> entry ;  |
                               <labellist> entry <parameterlist> ;  |
                               <labellist> entry <entry_optionslist> ;  |
                               <labellist> entry <parameterlist> <entry_optionslist> ;  ;

<entry_optionslist>  ::=
                               <entry_option>  |
                               <entry_optionslist> <entry_option>  |

<entry_option>  ::=
                               returns ( <function_attribute_list> )  |
/*      Statements      */

<statement> ::=
                        <prefixlist> <statement_2> |
                        <statement_2> ;

<statement_2> ::=
                        <labellist> <statement_1> |
                        <statement_1> ;

<statement_1> ::=
                        <unconditional_statement> |
                        <conditional_statement> ;

<prefixlist> ::=
                        ( <prefix_string> ) : ;

<prefix_string> ::=
                        <prefix_element> |
                        <prefix_string> , <prefix_element> ;

<prefix_element> ::=
                        <prefix> |
                        <noprefix> ;

<prefix> ::=
                        conversion | fixedoverflow | overflow | size | subscriptrange |
                        stringrange | underflow | zerodivide | stringsize ;

<noprefix> ::=
                        noconversion | nofixedoverflow | nooverflow | nosize | nosubscriptrange |
                        nostringrange | nounderflow | nozerodivide | nostringsize ;


<labellist> ::=
                        <labellist> <reference> : |
                        <reference> : ;

<unconditional_statement> ::=
                        <block> |
                        <do_group> |
                        <goto_statement> |
                        <call_statement> |
                        <return_statement> |
                        <assignment_statement> |
                        <on_statement> |
                        <revert_statement> |
                        <io_statement> ;
```

Reproduced from best available copy.

```
/*        Compound__Statements         */

<block>  ::=
                      begin ; <body>  !

<do_group>  ::=
                      <simple_group>  !
                      <iterated_group>  !

<simple_group>  ::=
                      do ; <end_clause>  !
                      do ; <sentence_list> <end_clause>  !

<iterated_group>  ::=
                      do <do_specification> ; <body>  !
                      do while ( <expression> ) ; <body>  !

<do_specification>  ::=
                      <reference> = <specification_list>  !

<specification_list>  ::=
                      <specification>  !
                      <specification_list> , <specification>  !

<specification>  ::=
                      <expression>  !
                      <expression> by <expression>  !
                      <expression> by <expression> to <expression>  !
                      <expression> to <expression>  !
                      <expression> to <expression> by <expression>  !
                      <expression> while ( <expression> )  !
```

```
/*        Flow  of  Control  Statements          */

<conditional_statement>  ::=
                                   <if_clause> <statement>  |
                                   <if_clause> <balanced_statement> else <statement>  !

<if_clause>  ::=
                                   if <expression> then  !

<balanced_statement>  ::=
                                   <if_clause> <balanced_statement> else <balanced_statement>  |
                                   <unconditional_statement>  !

<goto_statement>  ::=
                                   goto <reference> ;  |
                                   go to <reference> ;  !

<call_statement>  ::=
                                   call <reference> ;  !

<return_statement>  ::=
                                   return ;  |
                                   return ( <expression> ) ;  !
```

```
/*        Storage__Manipulation        */

<assignment_statement>  ::=
                            <reference_list> = <expression> ;  !

<reference_list>  ::=
                            <reference>  !
                            <reference_list> , <reference>  !
```

```
/*        Condition  Handling        */

<on_statement>  ::=
                          on <condition_list> <unconditional_statement>  |
                          on <condition_list> system ;  |

<condition_list>  ::=
                          <condition>  |
                          <condition_list> , <condition>  |

<revert_statement>  ::=
                          revert <condition> ;  |

<condition>  ::=
                          <prefix>  |
                          <io_condition>  |
                          error  |

<io_condition>  ::=
                          <io_condition_key> ( <file_expression> )  |

<io_condition_key>  ::=
                          endfile  |  endpage  |  key  |  record  |  transmit  |  undefined  ;
```

```
/*      Declarations       */

<declaration_sentence>  ::=
                               declare <declaration_list> ;  |

<declaration_list>  ::=
                               <declaration>  |
                               <declaration_list> , <declaration>  |

<declaration>  ::=
                               <integer> <declaration_3>  |
                               <declaration_3>  |

<declaration_3>  ::=
                               <declaration_2> <attribute_list>  |
                               <declaration_2>  |

<declaration_2>  ::=
                               <declaration_1> <dimension_attribute>  |
                               <declaration_1>  |

<declaration_1>  ::=
                               <identifier>  |
                               ( <declaration_list> )  |

<dimension_attribute>  ::=
                               ( <bound_pair_list> )  |

<bound_pair_list>  ::=
                               <bound_pair>  |
                               <bound_pair_list> , <bound_pair>  |

<bound_pair>  ::=
                               <expression>  |
                               <expression> : <expression>  |
                               *  |
/*        Attributes          */


<attribute>  ::=
                              <data_attribute>  |
                              <non_data_attribute>  |
                              <scope_attribute>  |

<data_attribute>  ::=
                              <arithmetic_attribute>  |
                              <string_attribute>  |
                              <storage_class_attribute>  |
                              <initial_attribute>  |
                              picture  |  pointer  |  label  |  varying  |  aligned  |  unaligned

<arithmetic_attribute>  ::=
                              <arithmetic_attribute_key> ( <integer> )  |
                              <arithmetic_attribute_key> ( <integer> , <signed_integer> )  |
                              <arithmetic_attribute_key>  |

<arithmetic_attribute_key>  ::=
                              decimal  |  binary  |  fixed  |  float  |  real  |


<signed_integer>  ::=
                              + <integer>  |
                              - <integer>  |

<string_attribute>  ::=
                              <string_attribute_key> ( <expression> )  |
                              <string_attribute_key> ( * )  |

<string_attribute_key>  ::=
                              bit  |  character  |


<storage_class_attribute>  ::=
                              based ( <basic_reference> )  |
                              automatic  |
                              static  |
                              based  |

<initial_attribute>  ::=
                              initial <initial_itemlist>  |

<initial_itemlist>  ::=
                              ( <initial_item_string> )  |
<initial_item_string>  ::=       <initial_item>  |
                                  <initial_item_string> , <initial_item>  |

<initial_item>  ::=               <initial_item_key> <constant>  |
                                  <initial_item_key> <reference>  |
                                  <constant>  |
                                  <reference>  |
                                  ( <expression> )  |
                                  <initial_iteration>  |
                                  *  |

<initial_item_key>  ::=           +  |  -  |  ^  |

<initial_iteration>  ::=          ( <expression> ) <initial_itemlist>  |
                                  ( <expression> ) *  |
                                  ( <expression> ) <initial_item_key> <simple_string_constant>  |
                                  ( <expression> ) <initial_item_key> <real_constant>  |
                                  ( <expression> ) <initial_item_key> <reference>  |

<non_data_attribute>  ::=         <entry_name_attribute>  |
                                  <file_attribute>  |
                                  builtin  |

<entry_name_attribute>  ::=       entry ( <descriptor_list> )  |
                                  returns ( <function_attribute_list> )  |
                                  entry  |

<descriptor_list>  ::=            <descriptor>  |
                                  <descriptor_list> , <descriptor>  |

<descriptor>  ::=                 <integer> <dimension_attribute> <attribute_list>  |
                                  <integer> <dimension_attribute>  |
                                  <integer> <attribute_list>  |
                                  <dimension_attribute> <attribute_list>  |
                                  <integer>  |
                                  <dimension_attribute>  |
                                  <attribute_list>  |

<attribute_list>  ::=
                                   <attribute>  |
                                   <attribute_list> <attribute>  ;

<function_attribute>  ::=          <data_attribute>  ;

<file_attribute>  ::=              file  |  environment  |  stream  |  record  |  input  |  output  |  update  |
                                   sequential  |  direct  |  keyed  |  print  |  backwards  ;

<scope_attribute>  ::=             internal  |  external  ;
```

```
/*        Input/Output  Statements        */

<format_sentence>  ::=
                         <labellist> format  |

<io_statement>  ::=
                         <open_statement>  |
                         <close_statement>  |
                         <stream_io_statement>  |
                         <record_io_statement>  |

<open_statement>  ::=
                         open <open_option_string> ;  |

<open_option_string>  ::=
                         <open_optionslist>  |
                         <open_option_string> , <open_optionslist>  |

<open_optionslist>  ::=
                         file ( <file_expression> ) <open_file_info_list>  |
                         file ( <file_expression> )  |

<open_file_info_list>  ::=
                         <open_file_info>  |
                         <open_file_info_list> <open_file_info>  |

<open_file_info>  ::=
                         <file_attribute>  |
                         ident ( <expression> )  |
                         title ( <expression> )  |
                         linesize ( <expression> )  |
                         pagesize ( <expression> )  |

<file_expression>  ::=
                         <unqualified_reference>  |

<close_statement>  ::=
                         close <close_option_string> ;  |

<close_option_string>  ::=
                         <close_optionlist>  |
                         <close_option_string> , <close_optionlist>  |

<close_optionlist>  ::=
                         file ( <file_expression> ) <close_file_info_list>  |
                         file ( <file_expression> )  |
<close_file_info_list>  ::=
                              <close_file_info>  |
                              <close_file_info_list> <close_file_info>  |

<close_file_info>  ::=
                              ident ( <expression> )  |
                              environment  |

<stream_io_statement>  ::=
                              get <stream_optionslist> ;  |
                              put <stream_optionslist> ;  |

<stream_optionslist>  ::=
                              <stream_file_info>  |
                              <stream_optionslist> <stream_file_info>  |

<stream_file_info>  ::=
                              file ( <file_expression> )  |
                              string ( <reference> )  |
                              <data_specification>  |
                              copy ( <file_expression> )  |
                              skip ( <expression> )  |
                              line ( <expression> )  |
                              column ( <expression> )  |
                              copy  (  skip  |  page  |

<data_specification>  ::=
                              <data_directed>  |
                              <edit_directed>  |
                              <list_directed>  |

<data_directed>  ::=
                              data ( <datalist> )  |

<edit_directed>  ::=
                              edit <datalist_string>  |

<datalist_string>  ::=
                              ( <datalist> )  |
                              <datalist_string> ( <datalist> )  |

<list_directed>  ::=
                              list ( <datalist> )  |

<datalist>  ::=
                              <datalist_element>  |
                              <datalist> , <datalist_element>  |
<datalist_element>  ::=          ( do <datalist> : <do_specification> )  |
                                  <expression>  ;

<record_io_statement>  ::=        <read_statement>  |
                                  <write_statement>  |
                                  <rewrite_statement>  |
                                  <locate_statement>  |
                                  <delete_statement>  |

<read_statement>  ::=             read file ( <file_expression> ) <read_info_list> ;  |

<read_info_list>  ::=             <read_info>  |
                                  <read_info_list> <read_info>  |

<read_info>  ::=                  into ( <reference> )  |
                                  set ( <reference> )  |
                                  ignore ( <expression> )  |
                                  key ( <expression> )  |
                                  keyto ( <reference> )  |

<write_statement>  ::=            write file ( <file_expression> ) from ( <reference> ) keyfrom ( <expression> ) ;  |
                                  write file ( <file_expression> ) from ( <reference> ) ;  |

<rewrite_statement>  ::=          rewrite file ( <file_expression> ) <rewrite_statement_1> ;  |
                                  rewrite file ( <file_expression> ) ;  |

<rewrite_statement_1>  ::=        key ( <expression> ) from ( <reference> )  |
                                  key ( <expression> )  |
                                  from ( <reference> ) key ( <expression> )  |
                                  from ( <reference> )  |

<locate_statement>  ::=           locate <unsubscripted_reference> file ( <file_expression> ) <locate_statement_1> ;  |
                                  locate <unsubscripted_reference> file ( <file_expression> ) ;  |

<locate_statement_1>  ::=         set ( <reference> ) keyfrom ( <expression> )  |
                                  set ( <reference> )  |
                                  keyfrom ( <expression> ) set ( <reference> )  |
                                  keyfrom ( <expression> )  ;
```

```
<delete_statement>  ::=
                          delete file ( <file_expression> ) key ( <expression> ) ;  |
                          delete file ( <file_expression> ) ;  !
```

```
/*      Expression        */

<expression>  ::=
                          <expression_6>  |
                          <expression> '| <expression_6>  !

<expression_6>  ::=
                          <expression_5>  |
                          <expression_6> & <expression_5>  !

<expression_5>  ::=
                          <expression_4>  |
                          <expression_5> <comparison> <expression_4>  !

<expression_4>  ::=
                          <expression_3>  !
                          <expression_4> '!'| <expression_3>  !

<expression_3>  ::=
                          <expression_2>  |
                          <expression_3> + <expression_2>  |
                          <expression_3> - <expression_2>  !

<expression_2>  ::=
                          <expression_1>  |
                          <expression_2> * <expression_1>  |
                          <expression_2> / <expression_1>  !

<expression_1>  ::=
                          <primitive_expression>  |
                          + <expression_1>  |
                          - <expression_1>  |
                          ^ <expression_1>  |
                          <primitive_expression> ** <expression_1>  !

<primitive_expression>  ::=
                          ( <expression> )  |
                          <reference>  |
                          <constant>  !

<comparison>  ::=
                          '<  |  '<=  |  =  |  '>=  |  ^=  |  ^'<  |  ^'>  !
```

```
/*        References  and  Constants        */

<reference>  ::=
                              <basic_reference>  |
                              <reference> -> <basic_reference>  |

<basic_reference>  ::=
                              <unqualified_reference_list>  |

<unqualified_reference_list>  ::=
                              <unqualified_reference>  |
                              <unqualified_reference_list> . <unqualified_reference>  |

<unqualified_reference>  ::=
                              <identifier>  |
                              <identifier> ( <expression_list> )  |

<expression_list>  ::=
                              <expression>  |
                              <expression_list> , <expression>  |
                              <expression_list> , *  |
                              *  |

<unsubscripted_reference>  ::=
                              <identifier>  |
                              <unsubscripted_reference> . <identifier>  |

<constant>  ::=
                              <real_constant>  |
                              <simple_string_constant>  |
                              <replicated_string_constant>  |

<real_constant>  ::=
                              <basic_real_constant>  |
                              <integer>  |
/*        The  Lexical  Grammar  of  PL/I          */


<lexical_non_terminal>  ::=
                              <identifier>  |
                              <integer>  |
                              <basic_real_constant>  |
                              <simple_string_constant>  |
                              <replicated_string_constant>  |

<non_lexical>  ::=
                              '040  |  '311  |  '012  |  '014  |

<identifier>  ::=
                              a->z  |
                              A->Z  |
                              <identifier> a->z  |
                              <identifier>  A->Z  |
                              <identifier>  _  |
                              <identifier>  0->9  |

<integer>  ::=
                              0->9  |
                              <integer>  0->9  |

<basic_real_constant>  ::=
                              <decimal_fixed_constant> b  |
                              <float_constant> b  |
                              <integer> b  |
                              <decimal_fixed_constant>  |
                              <float_constant>  |

<decimal_fixed_constant>  ::=
                              <integer> . <integer>  |
                              <integer> .  |
                              . <integer>  |

<float_constant>  ::=
                              <decimal_fixed_constant> e + <integer>  |
                              <decimal_fixed_constant> e - <integer>  |
                              <decimal_fixed_constant> e <integer>  |
                              <integer> e + <integer>  |
                              <integer> e - <integer>  |
                              <integer> e <integer>  |

<simple_string_constant>  ::=
                              <bit_string>  |
                              <character_string>  |

<bit_string>  ::=
```

Preceding page blank

```
                                  <character_string> b  !

<character_string>  ::=
                                  " <any_string> "  !

<replicated_string_constant>  ::=
                                  ( <integer> ) <simple_string_constant>  !
```

Appendix E

LIS Application: express

E.1 Introduction

The express language was developed by Interactive Planning Systems, Incorporated as an interactive command language for their financial planning system. In this appendix, we introduce the Interactive Planning System (IPS) and describe the way in which the express language is utilized in the development of financial planning models. We then discuss the application of LIS to the development of an express processor, and conclude the appendix with an express console session.

E.2 An Introduction to IPS

In this section, we give a brief introduction to the Interactive Planning System, taking our discussion from the Interactive Planning System User Guide (IPS 72).

What's wrong with existing systems?

In the last four years online systems have become available to manipulate and analyze management data. These systems, usually offered by timesharing companies, have several characteristics:

a. Users buy the time to run the packages -- but they are not guaranteed results.

b. Users pay high prices -- but get limited support.

c. Users don't know what a run will cost before it begins -- but must pay for it anyway.

d. Basic capabilities are lacking. No system has had available the four components of a management decision support system:

   i. Model Builder
   ii. Statistical analyzer
   iii. Data base manager
   iv. Report generator

<u>What is IPS?</u>

The IPS system has four components:

a. <u>A Model builder</u>
   To let the user quickly and economically build and test new models.

b. <u>An analysis system</u>
   Multivariate and stepwise regression plus tests of significance, analysis of variance, data transformation, plots, and search capabilities.

c. <u>A data management system</u>
   To enter and edit data, to establish and modify the users protection system for models and data, to store plots, scatter-diagrams and models when necessary.

d. <u>A report generator</u>
   An easy way to build complex reports - even reports which include a combination of graphics and tabular data.

E.3 The express Language

The following description of the express language is obtained by typing "help;" when running IPS under the express processor.

The express language allows you to execute IPS without going through the normal question and answer dialogue. The purpose of this note is to acquaint you with the general character of the express language.

Examples:

```
perform model Dave using data A1 and A2 for 6 periods output
the results thru report Ness reporting periods from 2 to 6;
print model Dave and Ness;
update model dave run 2 with model ness run 3;
output model Dave run 7 thru report Ness;
```

The commands are made up of basic commands, model data and report specifiers, qualifiers and noise words. Noise words are simply disregarded by the system and are allowed for ease of expression only. Anything which is not understood is regarded as "noise".

---Perform Command

```
perform <model spec> <length spec> <data spec> <report spec>
<destination spec> ;
```

This command tells the system to run the indicated model, using the indicated data, and to produce a report using the indicated report to be output to the indicated destination. If the <model spec> doesn't specify a run number, then a new run will be made.

---Output Only Command

```
<model spec> <report spec> <destination spec> ;
```

If no command is found on the line but a <model spec> is given which includes a run number of an existing run, then that run is output according to the report specified.

---Pack Command

```
pack <model spec> <data spec> <report spec> ;
pack all models ;
pack all reports ;
pack all data ;
pack all files ;
```

The pack command causes all of the files associated with a given model, data file, or report to be compressed and stored away. This saves a substantial amount of disk storage. The express system will automatically unpack any files that the user refers to in any express command. Unpacking will take a little while to perform, but the economical use of storage makes this often worthwhile. The pack all forms operate as their name implies, to pack all files (with files) or all files of some certain class (as in all models).

---Update Command
update <data spec> with <data spec> ;
The update command will cause the contents of the first <data spec> to be updated with the contents of the second <data spec>.

---Finish Command
end ;
done ;
finish ;
Terminates express and returns control to "model data or analyze".

---Error Command
error ;
Causes a descriptive message concerning the last error which has occurred to be typed on the console.

---Help Command
help ;
Types (this) descriptive text.

---<model spec>
model <name>
model <name> run <n>
run <n> model <name>
These are alternative forms for the <model spec>. The first specifies a model only (the system will generate a new run number if it is needed). The others specify a particular model and run.

---<data spec>
data <name>
no data
The first of these specifies the data file name. The second specifies that no data is to be used (this is different from not mentioning the data in that the system requires a data

spec for making a run).

---<report spec>
report <name>
report <name> <report qualifier>
This specifies a particular report file.

---<report qualifier>
from <n> to <n>
from <n>
to <n>
The from qualifier gives the desired beginning period of the report. It is assumed to be 1 if no from is given. The to qualifier gives the desired ending period of the report. It is assumed to be the last period in the run if no to qualifier is given.

---<length spec>
for <n>
This specifies a run of <n> periods.

---<destination spec>
onto tty
onto teletype
on tty
on teletype
into name
The first four of these specify that the report is to be typed on the teletype. At the moment you might as well not give them because that is what will be assumed if no "into" appears. Later we will allow you to change the default width of your terminal using this specification.
The into specification causes the report to be filed automatically as report name.

---General Structures
Where it makes sense to specify more than one item (for example in a data specification in a perform statement), you may do that by separating items by the word "and". For example:

data Dave data Ness data answer

could be replaced by:

data Dave and Ness and answer

The break character "," (wherever it appears) is equivalent to the word "and". Thus the above example is identical to:

data Dave, Ness and answer

(Note: not data Dave, Ness, and answer as this equals data Dave and Ness and and answer).
The order of specification within a command doesn't matte much. Thus you could say perform writing of report Ness for 6 periods from 2 using data A1, A2 with model Dave; (which is equivalent to the first example in this note).

E.4 The express Processor

The express processor developed using LIS is not the actual express processor employed in IPS. IPS was developed quite independently of LIS, and the purpose of the present application is simply to illustrate the utilization of LIS in the development of interactive languages for management information systems. This being the case, the semantics of the language is limited to the simple manipulation and display of express command constructs. The LIS Language Definition of express is given at the end of this appendix.

There are two significant differences between the structure of the parsers (lexical and primary) produced for the express language and the parsers produced for the languages developed in the previous appendices. First, the parsers for express are interactive whereas the previous parsers were non-interactive. This is a minor implementation problem, and simply involves the implementation of a procedure for reading express command lines and activating the parsers.

The second difference is more interesting and involves handling the requirement for "noise" words in the express commands. Whereas the acceptance of "noise" words lends much flexibility to the express dialogue and makes possible the specification of express commands that look very much like English sentences, this flexibility is not without cost. The cost is associated with the severe restrictions placed on the ability to detect and report syntax errors, with the accompanying risk of accepting ambiguous commands. This, however, is true regardless of the parsing strategy employed, and is apparantly a language tradeoff that the designers of express were willing to accept. The actual implementation of the "noise" words required two modifications to LIS, one to the Processor Generator and the other to LIS Processor Control. The modification to the Processor Generator eliminates the computation of default look-ahead transitions. This is necessary in order to maintain deterministic parsing. The modification to LIS Processor Control is such that, when the current input symbol matches none of the transitions from the current read state or look-ahead state, the control procedure fetches the next input symbol rather than invoking the error handling facility.

E.5 express Console Session

The following is an express console session. The express processor is invoked by typing "express" at Multics command level. The express processor signals that it is ready to accept the next command sequence by typing ":". Multiple commands are allowed per command sequence, and command sequences may extend over several lines. The end of an express command sequence is indicated by two carriage returns. The express console session:

```
express

:perform model division;


          Perform Command:
          <model spec>
               division

:perform model division onto tty using data east and west
data south for 6 periods output the results thru report
consolidated reporting periods from 3 to Z;

The last of the above commands cannot be recognized,
please reissue.

 error;


          Error Command:
          error

:perform model division onto tty using data east and west
data south for 6 periods output the results thru report
consolidated reporting periods from 3 to 6;


          Perform Command:
          <model spec>
               division
          <destination spec>
               tty
          <data spec>
               east
               west
          <data spec>
               south
          <length spec>
               6
          <report spec>
               consolidated
               from 3
               to 6
```

:output the results of model division run 7 thru report consolidated:

```
          Output Command:
          <model spec>
               division
          <model spec>
               7
          <report spec>
               consolidated
```

:update data east and west with data north and south:

```
          Update Command:
          <data spec>
               east
               west
          <data spec>
               north
               south
```

:pack model industry data aggregate report final:

```
          Pack Command:
          <model spec>
               industry
          <data spec>
               aggregate
          <report spec>
               final
```

:perform model industry run 5 data aggregate for 6

```
report industry from 1962 to 1973 into conclusions:
pack model industry data aggregate report industry:


           Perform Command:
           <model spec>
                industry
           <model spec>
                5
           <data spec>
                aggregate
           <length spec>
                6
           <report spec>
                industry
                from 1962
                to 1973
           <destination spec>
                conclusions

           Pack Command:
           <model spec>
                industry
           <data spec>
                aggregate
           <report spec>
                industry
```

:Activate IPS to perform the model industry using run 5 with data aggregate for 6 periods report industry results from 1962 to 1973 place results into conclusions file: pack the model industry with data aggregate report industry:

```
           Perform Command:
           <model spec>
                industry
           <model spec>
                5
           <data spec>
                aggregate
           <length spec>
                6
           <report spec>
                industry
                from 1962
                to 1973
           <destination spec>
```

```
          conclusions

     Pack Command:
     <model spec>
          industry
     <data spec>
          aggregate
     <report spec>
          industry

:finish the current activation of IPS:


     Finish Command:
     finish
```

```
dcl       1         text_reference_stack(top)     aligned   based(text_reference_stack_ptr),
                    2         construct            char(15)  aligned,
          text_reference_stack_ptr                 pointer   external  static,
          top                                      fixed     binary(35)          external  static,

          enter_spec                    entry(char(20) aligned, bit(1) aligned) internal,
          express_command_reduction     bit(1)    aligned   external  static,
          finish_command_reduction      bit(1)    aligned   external  static,
          ioa_                          entry     external,
          n_specs                       fixed     binary(17)          internal  static,
          print_spec_list               entry     internal,
          report_qualifier_1            char(15)  aligned   internal  static,
          report_qualifier_2            char(15)  aligned   internal  static,
          runoff                        entry     external,
          spec_list(100)                char(20)  aligned   internal  static,
          spec_list_break(100)          bit(1)    aligned   internal  static;


enter_spec:  proc(spec_entry, spec_entry_break);   /*                        *     enter_spec     *
                                                                             **********************
        This procedure enters specs into the spec_list.   */


        dcl       spec_entry          char(20)  aligned,
                  spec_entry_break    bit(1)    aligned;


        n_specs = n_specs + 1;
        spec_list(n_specs) = spec_entry;
        spec_list_break(n_specs) = spec_entry_break;
        return;
        end;


print_spec_list:  proc;   /*                                                 *     print_spec_list     *
                                                                             ***************************
        This procedure prints the spec_list.   */

        dcl       i                   fixed     binary(17)          internal  static;
```

```
          do i = 1 to n_specs;
                    if spec_list_break(i)
                       then call ioa_("^-^a", spec_list(i));
                       else call ioa_("^-     ^a", spec_list(i));
          end;
          n_specs = 0;
          return;
          end;


/*        Initialization Semantics          */

          finish_command_reduction = "0"b;
          n_specs = 0;
          return;


<lexical_non_terminal>  ::=
                              <identifier>  :
                              <integer>  !


<identifier>  ::=
                              <identifier> a->z  :
                              <identifier> A->Z  :
                              <identifier> 0->9  :
                              a->z  :
                              A->Z  !


<integer>  ::=
                              <integer> 0->9  :
                              0->9  !



/*        The EXPRESS Language          */

<primary_non_terminal>  ::=
                              <express_language>  !

/*        no semantics          */
          return;

<express_language>  ::=
                              <express_command_list>  !

/*        no semantics          */
          return;
```

```
<express_command_list>  ::=    <exprass_commend_list> <exprass_commend>  :
                               <express_command>  !

          express_command_reduction = "1"b;
          return;

<express_command>  ::=         <perform_command>  :
                               <output_command>  :
                               <pack_command>  :
                               <update_command>  :
                               <error_command>  :
                               <halp_command>  :
                               <finish_command>  !

/*        no semantics         */
          return;

<perform_command>  ::=         parform <parform_spac_list> ;  !

          call ioa_("^/^-Parform Command:");
          call print_spec_list;
          raturn;

<perform_spec_list>  ::=       <parform_spec_list> <perform_spec>  :
                               <perform_spac>  !

/*        no semantics         */
          return;

<perform_spac>  ::=            <model_spec>  :
                               <length_spec>  :
                               <data_spac>  :
                               <raport_spac>  :
                               <destinetion_spec>  !

/*        no semantics         */
          raturn;

<output_command>  ::=          <output_spec_list> ;  !

          call ioa_("^/^-Output Commend:");
          cell print_spec_list;
          return;
```

```
<output_spec_list>  ::=
                              <output_spec_list> <output_spec>  :
                              <output_spec>  !

/*        no semantics        */
          return;


<output_spec>  ::=
                              <model_spec>  :
                              <report_spec>  :
                              <destination_spec>  !

/*        no semantics        */
          return;


<pack_command>  ::=
                              pack <pack_spec_list> ;  :
                              pack all models ;  :
                              pack all reports ;  :
                              pack all data ;  :
                              pack all files ;  !

          call ioa_("^/^-Pack Command:");
          if alternative_number = 1
             then call print_spec_list;
             else call ioa_("^-^a ^a", construct(top - 2), construct(top - 1));
          return;


<pack_spec_list>  ::=
                              <pack_spec_list> <pack_spec>  :
                              <pack_spec>  !

/*        no semantics        */
          return;


<pack_spec>  ::=
                              <model_spec>  :
                              <date_spec>  :
                              <report_spec>  !

/*        no semantics        */
          return;


<update_command>  ::=
                              update <data_spec> with <data_spec> ;  !

          call ioa_("^/^-Update Command:");
          call print_spec_list;
    -     return;
```

```
<error_command>  ::=
                              error ;  !
         call ioa_("^/^-Error Command:^/^-arror");
         return;


<help_command>  ::=
                              help ;  !

         call runoff("exprass.halp");
         return;


<finish_command>  ::=
                              end ;  :
                              done ;  :
                              finish ;  !

         finish_command_reduction = "1"b;
         call ioa_("^/^-Finish Command:");
         call ioa_("^-^a^3/", construct(top - 1));
         raturn;


<model_spac>  ::=
                              <modal_spac> <and> <identifier>  :
                              <model_spac> <and> <integer>  :
                              model <identifier>  :
                              run <integar>  !

         if alternetive_number > 2
            then call enter_spec("<model spac>", "1"b);
         call enter_spac(construct(top), "0"b);
         raturn;


<deta_spec>  ::=
                              <data_spec> <and> <identifier>  :
                              <data_spec> <and> no data  :
                              data <identifier>  :
                              no data  !

         if alternative_numbar > 2
            then call enter_spec("<data spec>", "1"b);
         if elternative_number = 1  :  alternetiva_number = 3
            then call enter_spec(construct(top), "0"b);
            else cell enter_spec("no deta", "0"b);
         return;


<report_spec>  ::=
                              <report_spec> <end> <report_quelifier>  :
                              <report_spec> <end> <identifier>  :
                              report <identifier> <report_quelifier>  :
                              report <identifier>  !
```

```
        if alternative_number > 2
           then call enter_spec("<report spec>", "1"b);
        if alternative_number = 3
           then call enter_spec(construct(top - 1), "0"b);
        if alternative_number = 1  :  alternative_number = 3
           then do;
                 call enter_spec(report_qualifier_1, "0"b);
                 if report_qualifier_2 ^= " "
                    then call enter_spec(report_qualifier_2, "0"b);
           end;
        if alternative_number = 2  :  alternative_number = 4
           then call enter_spec(construct(top), "0"b);
        return;


<report_qualifier>  ::=         from <integer> to <integer>  :
                                to <integer> from <integer>  :
                                from <integer>  :
                                to <integer>  !

        report_qualifier_2 = " ";
        if alternative_number = 1
           then do;
                 report_qualifier_1 = "from "::substr(construct(top - 2), 1, index(construct(top - 2), " ") - 1);
                 report_qualifier_2 = "to "::substr(construct(top), 1, index(construct(top), " ") - 1);
           end;
        if alternative_number = 2
           then do;
                 report_qualifier_1 = "to "::substr(construct(top - 2), 1, index(construct(top - 2), " ") - 1);
                 report_qualifier_2 = "from "::substr(construct(top), 1, index(construct(top), " ") - 1);
           end;
        if alternative_number = 3
           then report_qualifier_1 = "from "::construct(top);
        if alternative_number = 4
           then report_qualifier_1 = "to "::construct(top);
        return;


<length_spec>  ::=              <length_spec> <and> <integer>  :
                                for <integer>  !

        if alternative_number = 2
           then call enter_spec("<length spec>", "1"b);
        call enter_spec(construct(top), "0"b);
        return;


<destination_spec>  ::=         <destination_spec> <and> teletype  :
                                <destination_spec> <and> tty  :
                                <destination_spec> <and> <identifier>  :
                                onto teletype  :
                                onto tty  :
                                on teletype  :
                                on tty  :
                                into <identifier>  !
```

```
         if alternative_number > 3
            then call enter_spec("<destination spec>", "1"b);
         call enter_spec(construct(top), "0"b);
         return;

<and>  ::=
                              and  !
                              ,  !

/*       no semantics         */
         return;
```